Interface
2003 September

［表紙デザイン：㈱プランニング・ロケッツ］

Contents 2003 September

特 集

ソフトウェア技術者も必読！


```
#include "gray_code_counter.h"

void gray_code_counter::count_up(void)
{
  if ( RESET.read() ) {
    Y = 0x0;
  } else {
    Y = GRAY.read();
  }
};

void gray_code_counter::gray_code_gen(void)
{
  int i;
  sc_uint<F> TMP = Y.read();
  sc_uint<F> BIN;
  sc_uint<F> TMP_GRAY;

  BIN[F-1] = TMP[F-1];
  for (i=F-2;i>=0;i--) {
    BIN[i] = BIN[i+1] ^ TMP[i];
  }

  BIN++;

  TMP_GRAY[F-1] = BIN[F-1];
  for (i=F-2;i>=0;i--) {
    TMP_GRAY[i] = BIN[i+1] ^ BIN[i];
  }

  GRAY.write(TMP_GRAY);
};
```

Interface

Contents 2003 September

話題のテクノロジ解説

ショウレポート&コラム

一般解説&連載

情報のページ

連載「フリーソフトウェア徹底活用講座」,「開発技術者のためのアセンブラ入門」,「やり直しのための信号数学」,「音楽配信技術の最新動向」は，お休みさせていただきます．

インターネットセキュリティカンファレンス

RSA Conference 2003 Japan

北村俊之

米国では10年以上の歴史をもち，世界最大規模のデータセキュリティと暗号のカンファレンスとして知られる「RSA Conference 2003 Japan」が，6月3日(火)～4日(水)の2日間，東京国際フォーラムで開催された．主催は，RSA Conference 2003 Japan 実行委員会である．本カンファレンスは2002年5月の第1回開催に続き，日本では2回目の開催となる．

現在，日本のブロードバンドインターネット加入者は600万人を超え，日々増加し続けている．また，携帯電話をはじめとするモバイル端末からは，6000万人近くがインターネットに接続しているという．行政電子化の動きも本格化しており，インターネットは日常生活に欠かすことのできないインフラとして定着した感がある．反面，個人情報保護やネットワーク利用犯罪などへの対応が，深刻な課題となっている．

本カンファレンスでは，こうした情報セキュリティに不可欠な暗号技術をはじめ，インターネット，ワイヤレス，モバイルに関するセキュリティ全般，技術標準や法制面での動向など幅広い分野を網羅し，セキュリティ技術標準，情報セキュリティと危機管理など九つのテーマで，50以上の講座が開催された．6月3日には米国ホワイトハウス元特別補佐官リチャード・クラーク氏や，RSA暗号技術を開発したイスラエルのワイツマン研究所教授のアディ・シャミア氏の基調講演が行われた．

また，これらの各分野に関連した企業約30社による展示会も併催されるなど，セッション/展示会ともに大幅なスケールアップがはかられていた．

● 出展された製品/技術

RSAセキュリティは，企業のセキュリティ施策に不可欠な，アイデンティティ＆アクセスマネジメントをテーマに展示を行っていた．また，携帯電話を利用したユーザー認証やWebアクセス管理，暗号化ツール「RSA SecurID」などのデモを行っており，来場者の関心も高かった(**写真1**)．

〔**写真1**〕**RSA SecurID**

伊藤忠テクノサイエンスでは，Net Screen Technologies社の統合型セキュリティアプライアンス製品「Net Screen Security Appliance」をベースとした，メール系ポリシーマネジメントなどの展示を行っていた．同製品は，ファイアウォール，VPN，トラフィック管理を統合しており，セキュリティ専用のASIC技術の採用により，低遅延のIPSec高速暗号処理を特徴としている．また，多様なネットワーク環境へのシームレスな統合が可能であるという．

セキュアソフトは，IDS/IDPS/Firewall/VPN機能一体型のアプライアンス製品「SecureSoft Tシリーズ」の展示を行っていた．同製品は，SOHO(T-30，**写真2**)から大規模ネットワーク環境(T-1000)に対応するラインナップを備えており，高価な専用ワークステーションを別途用意する必要がないことが，大きな特徴とのことである．また，IDSのアプライアンスサーバを基本に，IDPS/Firewall/VPNの各機能は，必要に応じてオプションで選択できる．IPSec方式のVPNを提供しており，IPSec専用のVPNゲートウェイ装置としても機能し，IPSec標準をサポートする他のVPN機器との連携も可能である．

〔**写真2**〕**SecureSoft T-30**

テクマトリックスでは，Aventail社のSSL VPN製品「EX-1500」を中心に，Webアプリケーション監査/監視ソリューション，無線LANセキュリティ製品，ウィルス対策製品の展示を行っていた．

日本電気通信システムでは，FireWall，VPNおよびブロードバンドルータ機能を提供する「SecureBlade」に注目が集まっていた(**写真3**)．同製品は，集中管理ソフト「SMP」と組み合わせて導入することで，企業内の複数のリモートサイトにまたがるセキュリティレベルをローコストで容易に維持管理できるという．

〔**写真3**〕**日本電気通信システムのブース**

アラジン ジャパンは，PKIにおける電子証明書を安全に格納し，VPNでは2因子認証を提供，メールではメッセージの暗号化に対応したセキュリティトークン「eToken」で，来場者の注目を集めていた(**写真4**)．こちらは，USBポートに差し込むだけで，高いセキュリティとポータビリティを低価格で実現できるのが大きな特徴だという．

〔**写真4**〕**アラジン ジャパンのブース**

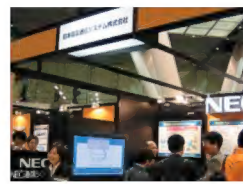
三菱電機は電子署名ソリューションを主体とした展示を行っており，なかでも「MistyGuard <SignedPDFファミリー>」は来場者の注目度が高い製品だった(**写真5**)．これはICカードを利用してPDFファイルへの電子署名および印影の表示ができるため，ペーパレスによる業務コストの大幅な削減が可能とのことだった．

〔**写真5**〕**Signed PDF**

日本ルーセント・テクノロジーでは，VPNファイアウォール「Brickファミリ」製品(**写真6**)を利用した，マネージドセキュリティ，PC/PDAからセキュアにアクセスするためのリモートアクセスVPN，および認証VLANの各ソリューションに関する展示を行っていた．

〔**写真6**〕**Brickファミリ**

シマンテックは，侵入検知，ポリシー監査，脆弱性検査，ファイアウォールをはじめとする，トータルセキュリティソリューションの展示を行っていた．とくに来場者の注目を集めていたのが「Symantec Gateway Security」である．これはファイアウォール，VPN，侵入検知，アンチウィルス，コンテンツフィルタリングをゲートウェイに統合することで，ウィルスやワーム，ネットワーク攻撃などの脅威から，ネットワークを包括的に保護することができるという．

京セラコミュニケーションシステムでは，脆弱性，セキュリティリスクのネットワーク型常時診断を実現したアプライアンス製品「nCircle/IP360」の展示，紹介が行われていた(**写真7**)．

〔**写真7**〕**nCircle**

Show&News Digest

TOPPERSプロジェクト結成

■日時：2003年6月24日（火）
■場所：アルカディア市谷（東京都千代田区）

オープンソース版ITRON「TOPPERS」の開発・普及を行うことを目的として「TOPPERSプロジェクト」が結成され，同日付けで会員募集を開始した．TOPPERSは，豊橋技術科学大学組込みリアルタイムシステム研究室（現在は名古屋大学へ移籍）の高田広章氏を中心に開発されたμITRON仕様準拠リアルタイムOSの名称である．

TOPPERSプロジェクトが発足した背景として，μITRONの「多様すぎる実装」が問題であるという認識があった．たとえばプロトコルスタック（TCP/IPプロトコルスタック，GUIスタックなど）は，特定のμITRON専用として開発・販売されていることがあり，相互に互換性がないという問題点も指摘されていた．そこでTOPEERSをμITRON実装系の「決定版」と位置づけ，互換性問題の解決法として提案していくことにした．

また，従来のオープンソースプロジェクトが潜在的に抱える問題として「品質保証」と「著作権や知的財産権を侵害する危険性」があった．このようなプロジェクトは誰でも自由に開発に参加することが可能なものが多く，開発期間の短縮に貢献してきたという実績がある．しかしその反面，「誰でも」参加できることから質の低いコードが混入する可能性もあり，また，それをチェックする機構もなかった．また，悪意の有無に関わらず，他人の著作権を侵害したコードが混入する可能性もあるという問題点が指摘されていた．とくに組み込み機器で用いられることの多いTOPPERSでは，これらは重要な問題となっていた．

そのためTOPPERSはプロジェクトでは，基本的に「公式リリース」はTOPPERSプロジェクト会員のみによって開発し，会員には「著作権侵害のない開発」を義務づけるという形でこれらのリスクを低減するという手法を採用した．公式リリースに関しては開発者を限定することになるため，コード品質も一定に保たれることや，違法なコードの混入を未然に防ぐことが期待できる．これは非会員の参加を拒むものではなく，通常の成果物から，一定の品質をもった「公式リリース」を作り出すための手法と位置づけられている．通常の開発は，一般参加者も含めてこれまでどおりに続けられる．

今後のTOPPERSプロジェクトのロードマップとしては，μITRONフルセットバージョンの開発，C++への対応，ダイナミックローディング機能（IDL）の開発，非対称マルチプロセッサへの対応，時間保護（タスクが使用できるCPU時間の保証）などが予定されている．

TOPPERSプロジェクトでは，2年後に全μITRON中の50%のシェア，4年後に80%のシェアを獲得することを目標としており，これを実現するために，半導体メーカーに対してメーカー製独自μITRONからTOPPERSへの乗り換えを働きかける，自社製μITRONを製作しているメーカーに対してTOPPERSへの乗り換えを働きかけることなどを行う．

TOPPERSプロジェクトの会長は名古屋大学教授の高田広章氏，副会長は宮城県産業技術総合センターの高橋賢一氏，（株）リコーの竹内良輔氏，（株）エーアイコーポレーションの加藤博之氏．TOPPERSの開発/保守/普及促進/教育などを活動内容とし，会費は年会費￥10万，入会費￥10万．現在東京都に対してNPO法人化を申請しており，夏頃をめどに認可予定とのことだった．

名古屋大学大学院教授
高田広章氏

TOPPERSのロゴマーク

アナログデバイセズ，TIGER SHARC DSP 3製品を発表

■日時：2003年6月19日（木）
■場所：アーバンネット大手町レベル21（東京都千代田区）

アナログ・デバイセズ（株）は，高性能アプリケーション向けのDSP，ADSP-TS201/TS202/TS203を発売した．同製品は最高600MHzで動作し（TS201），4,800MMACS/3,600MFLOPSの性能をもつほか，内部に最大24MビットのeDRAM（組み込みDRAM）を内蔵している．

内部的には128ビット幅のバスを4系統もち，バンド幅は38.4Gバイト/秒．内部バスにメモリが接続されているため高速な処理が可能なほか，メモリを外付けするよりも低い消費電力で動作可能，信頼性の向上などが期待できる．また，ワイヤレス基地局アプリケーションなどでのマルチプロセッサ環境も想定されている．

TIGER SHARCは，同社のDSPラインナップの中でハイエンドな用途向けと位置づけられ，普及価格帯のSHARC，低消費電力アプリケーション向けのBLACKfinとは棲み分けがなされる．

TIGER SHARC搭載
マルチプロセッサボード

ユビキタスIDセンター，ユビキタスID実証実験に向け，標準IDタグを認定

■日時：2003年6月23日（月）
■場所：YRPユビキタスネットワーキング研究所（東京都品川区）

RFIDの研究開発および標準化活動を行っているユビキタスIDセンターは，ユビキタスID技術の実証実験をこの夏から開始し，同時に実験に用いるためのIDタグ3種を標準IDタグとして認定した．

この実験は今年いっぱいの予定で，神奈川県横須賀市のよこすか葉山農協において行われる「食品トレーサビリティ実験」．この実験のために（株）日立製作所のミューチップ，凸版印刷（株）のT-Junction，YRPユビキタスネットワーキング研究所/東京大学坂村研究室/ルネサステクノロジのeTRON/16-AE45Xが，同規格の標準IDタグとして認定された．

通信の基盤技術やネットワーク機器が多数展示される

SUPERCOMM 2003 Atlanta Report

松本信幸

アメリカのジョージア州アトランタで，今年もSUPERCOMM 2003が6月1日～5日までの日程で開催されました．SUPERCOMMは通信関係の展示会で，おもに基盤技術やネットワーク機器の展示が行われます．SUPERCOMMに限らず，最近の展示会はテロや不況の影響からか出展企業，参加者などが減少傾向にあります．SUPER COMM 2003も昨年と比較して，出展者数，来場者数などが3割から4割の減少となっていました．

● 昨年との差

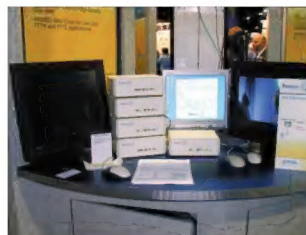
昨年は，曲げに強いシングルモード光ファイバや，モーダルバンド幅の値を向上させた高速伝送用マルチモード光ファイバ，そしてそれらの融着工具など，技術革新によるネットワーク基盤部分の出展が目を引きましたが，今年は，あまり新技術は見受けられず，従来技術によるソリューションの提供に関して，廉価版を市場に投入し始めているという感想をもちました．それらの中から，個人的に興味をもったものをいくつかを紹介します．

● アクセス系の状況

現在ではADSLがアクセス系の主流となっているようですが，ADSL関連機器に関しては，安価で，どちらかといえば家庭で使用する向きの(丸みを帯びた)デザインのものが見受けられました．しかし，速度向上のソリューションもやはりあり，ADSLの後継としてG.SHDSLの機器が出展されていました．

〔写真1〕Ethernet in the First Mile Alliance

それよりも気になるのは，FTTHのキーワードで知られている光ファイバによるアクセス関連です．アクセス系全般として「Ethernet in the First Mile Alliance(**写真1**，http://www.efmalliance.org/)という出展があったのですが，その展示内容はおもにGPONで，勧告の標準化が進められている状況が書かれていました(IEEE P802.3ah：1000Base-PX)．このGPONの製品群はADSLによるサービスを強く意識しているようで，展示していた製品について価格を問い合わせたところ，「ADSLより少々安く」という回答でした．ちなみに製品形状は，ADSLの製品群より二まわり位大きいものでした(**写真2**)．

この製品のインターフェースは1000Base-PXとして1000Base-X(1250Mbps/8B10B)系のもので，1本(一対ではない)の光ファイバに，上り回線用に1490nm帯のレーザ，下り回線用に1310nm帯のレーザを使用しています．中間に(パッシブ)スプリッタをもち，1:8接続を実現，上り回線は時分割(1回線あたり125Mbps相当)，下り回線は1Gbpsを8端末でシェアというものでした．日本企業では日立製作所が出展していました(Allianceブースではなく，自社ブース)．

〔写真2〕Passave Technologiesの PON

無線系のものとして面白いものに，屋外用のIEEE802.11bアンテナで，指向性をもたせ絞ったビームでスポットライトのようにホットスポットを作り出すというものがありました(**写真3**)．

〔写真3〕VIVATOの802.11b屋外用アンテナ

● インターネット電話の状況

インターネット電話関連でいちばん目に付いたのは，安価なIP電話機が登場してきていたことです．昨年までよく見かけられたファンクションキーの多くある機種も依然展示されていましたが，シンプルな形状のものも多く見受けられました(**写真4**，**写真5**)．確認できた限りでもっとも安価だったものは，SWISSVOICE(展示していたのはSylantro Systems(http://www.sylantro.com/)の$100を切るというものでした．

〔写真4〕カナダFlash Horizonの IP電話機

〔写真5〕韓国HS Teliannの IP電話機

画像をともなうマルチメディア通信として，音声のみではなくテレビ電話端末として出展されているものもありました．目を引いたものとしてはMarconi(http://www.marconi.com/)が出展していたものがあります(**写真6**)．音声がITU-T G.711，画像がMPEG-2で，最大五者通話が可能となっているという特徴があります．

〔写真6〕Marconiのテレビ電話端末

● 放送系アプリケーションの状況

画像を扱うものとして，放送系のアプリケーションがNTTから出展されていました(**写真7**)．従来までの，画像の綺麗さを見せる展示ではなく，家庭内でくつろいでテレビを見るような感じの展示内容となっていました．目玉はIGAP(Internet Group membership Authentification Protocol)という新しいプロトコルで，このプロトコルを利用することにより，従来まで困難だった課金を容易に行えるようにするとともに，契約内容などにおけるチャネル単位の受信の制限なども制御できるようになるということでした．これによってIPネットワーク上でケーブルテレビのようなサービスを容易に提供できるようになるというものです．

〔写真7〕NTTの画像配信サービス

＊　　＊

このほか，MPLS技術を利用し，Ethernet上に回線交換のサービスを実現するというのもあり，今後が楽しみではあります．次回2004年の会場は，Chicagoに移るそうです．

まつもと・のぶゆき　(株)タムラ製作所

移り気な情報工学 34

ユビキタスなエネルギー

山本 強

今，ユビキタス(Ubiquitous)が時代のキーワードになっている．ユビキタスとは，「どこにでもある」ということを意味する形容詞であり，ユビキタスが意味するところはユビキタスな物やサービスが素晴らしいということでもある．

目下のところ，いちばん注目を集めているユビキタスな物といえば，非接触ICタグであろう．1mm角以下のチップに128ビット程度のユニークなIDを記録し，それを近くにあるリーダから無線で読み取る仕掛け，つまりRFIDである．

理論もはっきりしていて，電波でICを動作させるための電力を送り，それを使って逆に情報を電波で送り返す仕組みである．電子工学系のエンジニアなら，原理的に可能なことは理解できるのだが，実現するのはそう簡単ではない．今のところ，この1mm角以下のRFIDは，単体では動作するために必要な電力を獲得できず，数cmの外部アンテナを接続して初めて情報交換が可能になる．そのため，この種の技術のエッセンスは，ICチップとアンテナの接続技術だったりする．

この例を見るまでもなく，ユビキタスな物は動作するための電力を獲得することが最大の問題になることが多いのである．

空間はエネルギーで満ちている

ユビキタスなIT機器は徹底的に低消費電力でなければならないから，動作に必要な電力も相当に小さくしなければならないのは当たり前である．たとえばソーラー電卓は，10平方センチメートル程度の太陽電池で蛍光灯程度の明るさでも計算してくれるが，この太陽電池の発電量は蛍光灯下では1mW程度のものである．そこまで大きくなくても，μWオーダの電力なら意外と簡単に手に入るのである．

たとえば，ゼーベック効果により，金属接合に温度差を与えることで電力が発生する．これは熱電対の動作原理でもあるが，問題は温度差がどこにあるかである．もし，人が身につけるものであれば，体温と外気温の差をエネルギーとして取り出すことができる．効率はおそろしく低いのだが，人間が食べた食料を電力に変換しているとも考えられる．

ものは試しで，実験室に転がっていたペルチェ冷却素子の片面を手に貼り付けてテスタで出力を計ってみたら40mV，40μA，つまり1.6μWの電力が連続して出ていることがわかった．したがって，理屈の上ではユビキタスなウェアなどというものが作れることになる．

ユビキタスなエネルギーは密度が低い

このように，エネルギーはいたるところに存在する．まさしくユビキタスなエネルギーである．しかし，ユビキタスなエネルギーは密度が低い．密度がいちばん高いと思われる太陽光エネルギーでも，取り出せるのは1平方メートルあたりたかだか100Wである．これを大きいと見ることもできるが，バッテリやガソリンといったパッケージ型エネルギーと比べると相当に密度が低いのである．

たとえば，100馬力のエンジンで走る自動車を考えてみよう．巡航時出力を25馬力とし，それで時速100kmで1時間，つまり100km走行するのに10ℓのガソリンを使うとする．これはリッターあたり10kmだから普通の感覚である．

電気屋は馬力という単位に慣れていないが，1馬力は約0.75kWに換算される．つまり，小型自動車のエンジンは10ℓで25馬力×0.75×1時間動くということになり，約19kWhのエネルギーを発生していることになる．このエネルギーを単一電池（1.5V 1Ah）に換算すると12,666個分というとんでもない量になる．単一電池1本を100円としても120万円以上となり，いかにガソリンのエネルギーが安く，密度が高いかがわかる．ちなみに19kWhの電力料金は約380円であり，10ℓのガソリン代1,000円とオーダが同じである．電気も石油から作られているということの証でもある．

ところで，25馬力相当の電力を太陽電池で発生させるには，どのくらいの面積が必要になるのだろうか．1平方メートルで100Wとするならば，190平方メートル，つまり13m×13mという自動車としては非現実的な面積の太陽電池が必要になる．かりに効率が2倍に上がってもまだ9m×9mの面積が必要だから，どんなにがんばっても実用となるソーラーカーはできないと断言してよい．

生命体というユビキタスエネルギー変換システム

生命体は，どこにでもある食料を勝手に食べてエネルギーを獲得しているのだから，本質的にユビキタスエネルギーを前提にしていることになる．動物は食料を消化してブドウ糖に変換し，血液を媒体として全身に配送する．つまり，血液を使って発電する仕組みができれば，人間と一体化して電力供給のいらない情報システムができることになる．現実に，燃料電池はそれに近い仕組みである．もし，血液を燃料として使う燃料電池ができたなら，それに血管を接続すると人間が食事をするとコンピュータが動きだすということが可能になる．これで一度体内に埋め込むと，生きている限り動き続けるユビキタスな情報機器ができることになる．

たしかに，映画「MATRIX」の世界はそんな世界である．

やまもと・つよし　北海道大学大学院工学研究科電子情報工学専攻
計算機情報通信工学講座　超集積計算システム工学分野

特集　ソフトウェア技術者も必読！

C/C++によるハードウェア設計入門

これまでハードウェア設計に使用されてきたHDLに代わり，C/C++言語をハードウェア設計へと適応することに注目が集まっている．これらハードウェア設計用C/C++言語は，従来のC/C++をベースとし，それらにハードウェア設計向けの記述を拡張したものとなっているため，すでにC/C++言語を修得済みのエンジニアにとっては理解しやすい．ためしに本特集に掲載されているソースリストを見ていただきたい．見慣れたC/C++言語に若干拡張がなされた形で，文法を知らなくてもなんとなく動作が想像できるのではないだろうか？

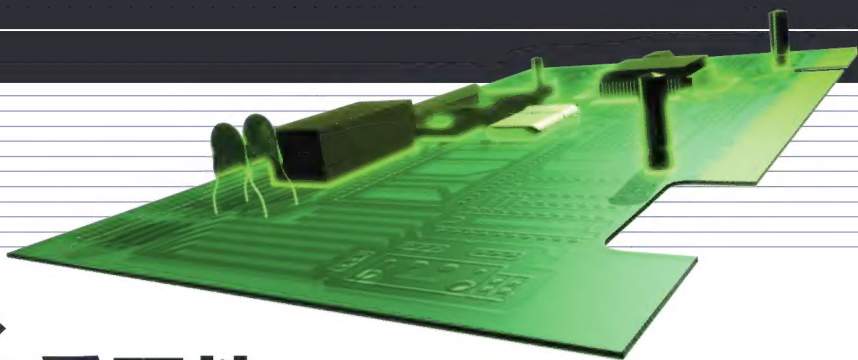
また，これらの言語は単にハードウェアのみの開発が目的ではない．ハードウェアとソフトウェアを別々に開発せず，システム全体として設計する「ハードウェア/ソフトウェア協調設計」が可能になるという大きな利点もある．これにより，ソフトウェアの不得手とする部分をハードウェア化するなどの最適化が容易になり，そのトレードオフの試行サイクルも大幅に短縮することができる．

そこで今回の特集では，システム記述言語SystemC/SpecCを用いて，実際にC/C++言語を使ったハードウェア設計法を修得する．

Prologue

新しい設計言語の時代へ

C/C++言語ベースのシステム設計の重要性

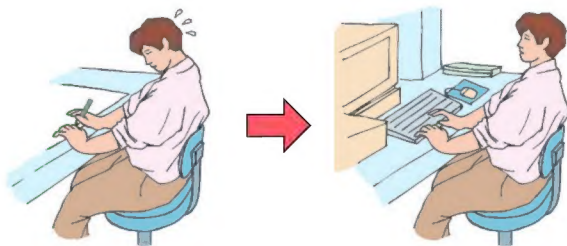
吉田たけお

1 半導体技術は進歩し続けている

LSI(Large Scale Integrated circuit：大規模集積回路)の1チップあたりに集積可能なトランジスタ数は，おおむねムーアの法則にしたがって増加しています．ムーアの法則とは，米Intel社の創設者の一人ゴードン・ムーア(Gordon E. Moore)が1965年に提唱した，「半導体の集積度は，1年半ごとに約2倍になる」という経験則のことをいいます．この法則によれば，約5年で10倍という驚異的なスピードで，集積規模が増加していくことになります．

単純に考えると，より高性能な半導体製品がより安価で手に入るわけですから，ユーザーの立場としては喜ばしいことです．しかし，同じく単純に考えると，実際に半導体製品を設計・製造する立場としては，設計すべきハードウェアの規模が約5年で10倍になるわけですから，生産性が高く実用的な設計手法の登場を期待することになります．

2 ハードウェア記述言語の登場

ほんの数年前までは，ハードウェア(ディジタル回路)の設計は，人手を使って回路図を書く，という作業によって行われてきました．実際には，ハードウェア設計を支援するCAD(Computer Aided Design)ツールを用い，ドローイングツールで絵を描くようにして，コンピュータ上で回路図を作成していました．しかし，これでは当然，設計生産性の向上は望めません．そこで登場したのが，ハードウェア記述言語(Hardware Description Language：HDL)です．

〔図1〕ハードウェア設計手法の変遷

HDLは，C言語やPascalなどのプログラミング言語と同様に，コンピュータ上で処理される形式言語(formal language)です．このHDLを用いることにより，プログラミング感覚でハードウェアの設計作業が行えます．HDLの登場により，設計作業は，机上(実際はコンピュータ上)で絵を描く作業から，コンピュータ上でプログラミングをする作業へと，移行してきました(**図1**)．現在では，HDLを抜きにしたハードウェア設計は，考えられなくなっています．

3 システム全体が設計対象になる

しかし半導体技術は進歩を続けており，これまで複数のICを用いなければ実現できなかった大きなシステムが，一つのICの中に実現できるようになってきました．このような大きなシステムを一つのICで実現することをシステムオンチップ(System on Chip：SoC)，SoCを実現したICをシステムLSI(System LSI)と呼んでいます．

ここでいうシステムには，ハードウェアだけでなく，その上で動作するソフトウェアも含まれています．すなわち，システムLSIの登場によって，設計の対象がハードウェアとソフトウェアを合わせたシステム全体に広がってきたことになります．

HDLの使用により，設計生産性が向上したことは事実ですが，ハードウェアのみを設計対象としたHDLでは，システム全体の設計を行うことはできません．そのため，システム全体の設計を可能とする設計環境が必要となってきたのです．

4 C/C++言語ベースのシステム設計

このような状況でC/C++言語が浮上してきたのは，当然の帰結といえます．HDLでシステム全体の設計を行えないのだから，プログラミング言語にハードウェアを設計する機能をもたせよう，という考え方です．このような，ハードウェアとソフトウェアの両方を設計できる言語をシステムレベル言語(system level language)といいます．そして，そのような役割をもたせるプログラミング言語としては，もっとも広く認知され，実際に用いられているC/C++言語となるわけです．

現在，このシステムレベル言語が注目を集めていますが，中でもC/C++言語をベースとした，SystemCとSpecCが有望株と

なっています．本特集では，SystemC や SpecC を用いた C/C++ 言語ベースのシステム設計について，その基本的な項目を解説します．とくに，これまで C 言語などによるプログラミングの経験はあるけれど，ハードウェアの設計には詳しくない，という方を対象とし，ハードウェアの基礎についても解説します．

〔図2〕各設計手法の違い

```
    cmpl %eax, -16(%ebp)
      jl.L21
    jmp .L19
    .p2align 2
.L21:
    movl -16(%ebp), %eax
    movl %eax, %eax
    leal 0(,%eax,4), %esi
    leal -584(%ebp), %eax
    movl %eax, %ebx
```

回路図(ハードウェア)を人手で作成

アセンブリ言語や機械語でソフトウェアを作成

(a) 回路図を人手で書いていた時代のハードウェアの設計

```
reg[ 1 : 0 ] CR_STATE;
wire[ 1 : 0 ] NX_STATE;

always @(negedge CK) begin
  CR_STATE <= NX_STATE;
end

assign Y = F_Y(CR_ST,X);

function[ 1 : 0 ] F_Y;
```

```
if(vdata.bi == BEND){
  access_on = TRUE;
  for(i=0;i<bb_inst;i++){
    k = i;
    for(j=i+1;j<bb+1;j++){
      if(b[j] < b_new[k]){
        k = j;
      }
    }
    if(k!=i){
```

ハードウェア記述言語(HDL)でハードウェアを設計

C/C++言語でソフトウェアを作成

(b) ハードウェア記述言語によるハードウェア設計

```
#include "systemc.h"

void st_machine::sm_reg(void)
{
  if ( RESET.read() ) {
    CR_ST = Q0;
  } else {
    CR_ST = NT_ST.read();
  }
};
```

C/C++言語ベースのシステムレベル言語でハードウェアとソフトウェアを同時に設計

(c) システムレベル言語によるシステム設計

5　システムレベル言語の効用

ところで，システムレベル言語を用いたシステム設計が，これまでの設計とどのように違うのか，見てみることにしましょう．

HDL の登場前は，**図2(a)**のような回路図を，人手で作成していました．このような回路図で表されるハードウェア上で動作するソフトウェアは，そのハードウェア専用のアセンブリ言語や機械語を用いてプログラミングします．また，専用の開発環境を用いて，C/C++ 言語などの高級言語によるプログラミングが行える場合もあります．いずれの場合も，ハードウェアの設計が完了しなければ，作成したソフトウェアの動作を正確に検証することはできません．

HDL 登場後は，**図2(b)**のように，ハードウェアを高級言語によるプログラミング感覚で設計します．これにより，ハードウェアの設計検証作業がかなり効率化されました．一方，ソフトウェアの設計に関しては，手書きの時代からあまり変化がありません．多くの場合，専用の開発環境を用いて，C/C++ 言語などの高級言語によるプログラミングが行われます．また，ソフトウェアの動作検証に関しても，ハードウェア設計の完了を待ってから行う必要があります．

以上の設計手法に対して，システムレベル言語を用いた設計手法では，**図2(c)**に示すように，まず，ハードウェアとその上で動作するソフトウェアから構成されるシステム全体をまとめて設計します．これをハードウェア/ソフトウェア協調設計といいます．その後，ハードウェア部とソフトウェア部を分割して，並行して，設計作業が進められます．ハードウェア部とソフトウェア部が同一の開発環境で設計できるので，両方の設計がある程度進んだ段階で，ハードウェア上でのソフトウェアの動作検証を効率的に行うことができます．これを，ハードウェア/ソフトウェア協調検証といいます．

このように，システムレベル言語を用いると，ハードウェア部とソフトウェア部を協調して，設計・検証が行えるため，設計期間の大幅な短縮につながると期待されています．

6　C/C++ 言語ベース設計に対するもう一つの期待

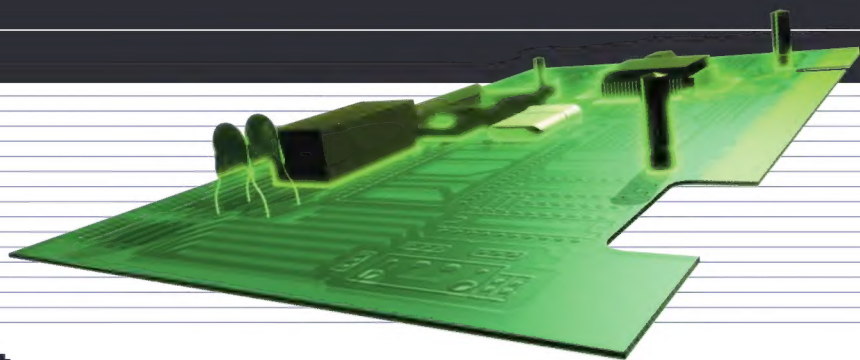
以上で概観したように，C/C++ 言語ベース設計は，今後のシステム設計における重要なカギを握っているといえます．さらに C/C++ 言語ベース設計には，もう一つの大きな期待があります．それは，ソフトウェア設計者をシステム設計者として取り込むことです．

ハードウェア設計者は，ソフトウェア設計者に比べると，その絶対数が圧倒的に少ないのが現状です．しかし，ハードウェア設計者の需要は増加の一途をたどっており，その人材不足が問題になっています．また，システムレベル設計ではハードウェア部とソフトウェア部を同時に設計するため，その設計者(システム設計者)には，ハードウェアとソフトウェアの両方の知識が要求されます．このようなシステム設計者も絶対的に不足しています．これらの人材不足をソフトウェア設計者で補うことによって解消したい，というのが C/C++ 言語ベース設計に対するもう一つの期待です．

実際，半導体メーカーの採用状況でも，ソフトウェア設計者の採用者数のほうが，ハードウェア設計者の採用者数を上回る傾向があります．ソフトウェア設計者がハードウェアやシステム全体を設計するような時代が来るのは，そんなに遠い未来のことではないのかもしれません．

よしだ・たけお　琉球大学 工学部 情報工学

Chapter 1

C++言語をベースにした システムレベル設計言語

SystemCの基礎

島尻寛之/吉田たけお

ハードウェアのみ，ソフトウェアのみではなく，「システム全体」を設計するという考え方が注目されている．システム全体を俯瞰(ふかん)することにより，ハードウェアで実現すべき部分とソフトウェアで実現すべき部分を適切に切り分け，パフォーマンスの高いシステムを設計することが可能になるだけでなく，消費電力などの面でも有利になる．そのためには，ハードもソフトも記述できる言語が必要になる．

SystemCは，数あるシステムレベル設計言語の中でもC++をベースとしているため，とくにソフトウェア技術者には親和性が高い．そのため，すでにC++を習得している技術者であれば，わずかな学習のみでシステムレベル設計が可能になる．

本章では，システムレベル設計言語SystemCの基礎を解説する．

（編集部）

はじめに

ハードウェア開発者だけでなく，ソフトウェア開発者でも**SystemC**という言語を耳にしたことがあると思います．このSystemCとは，どんな言語なのでしょうか？

SystemCは，大規模なシステムを効率良く設計するために開発された，C++言語をベースにしたシステムレベル設計言語です．大ざっぱには，ハードウェアも記述できるC++言語ということになります．SystemCはシステムレベル設計言語なので，システム上のハードウェアとソフトウェアを記述でき，さらには，システム全体の動作から，システム上の小さなディジタル回路の動作まで記述することができます．

このSystemCの登場により，多くのソフトウェア開発者がハードウェアの設計に携わるようになることは想像に難くありません．しかし，いままでソフトウェアの開発しか行ったことがない人にとっては，ハードウェアの設計は勝手が違い，とまどいを感じることと思われます．

そこで本特集の第1章～第4章では，「ハードウェアの設計をほとんど行ったことがない」，また「C言語は知っているけれどC++言語はあまり知らない」という方を対象に，SystemCによる基本的なハードウェア設計の手法について解説していきます．

まず，この章では，SystemCの基礎について説明します．

1 SystemCとは？

SystemCは，OSCI（Open SystemC Initiative）によって開発されているシステムレベル設計言語です．2000年4月に，最初のバージョンであるSystemC 1.0がリリースされ，2001年10月にSystemC 2.0がリリースされました．SystemCは現在もOSCIによって開発が進められており，2003年6月の時点でのバージョンは2.0.1となっています．また，OSCIからはSystemCのフリーのシミュレータもリリースされています．

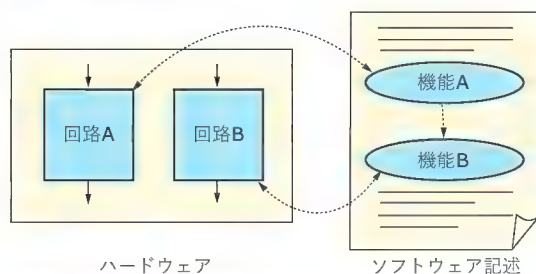
● システムとは？

SystemCが設計の対象としている「システム」とは，一体どういうものなのでしょうか？

航空券予約システム，航空管制システム，銀行の預金管理システムなど，「システム」という言葉はよく耳にします．このようなシステムは，スーパーコンピュータ，端末コンピュータ，大容量の記録装置，バックアップ装置などの数々のハードウェアから構成されています．そして，これらのハードウェア上でOSや種々のプログラムが稼働することによって，目的の処理が行われることになります．したがって，ハードウェアとソフトウェア全体でシステムの機能を実現していることになります．すなわちシステムとは，ある機能（処理）を実現するハードウェアとソフトウェアの集合ということになります．システム設計では，目的の機能を実現するハードウェアとその上で稼働するソフトウェアを設計・開発することになります．

それではなぜ，システム設計が重要視されるようになったのでしょうか？

近年，身のまわりにはカメラ付き携帯電話やPDA，ノートPCなどの小型で高性能なコンピュータがあふれています．そして，これら小型のコンピュータは，数年前の（比較的大きな）コンピュータと同等かそれ以上の性能をもっています．その一方で，サイズ，消費電力，価格は大幅に小さくなっています．これを実現できた大きな理由の一つは，コンピュータを構成する多くの部品が一つのチップに集積できるようになったことが

〔図1〕並列動作の記述

並列に動作する回路Aと回路Bに対して，プログラミング言語では回路Aと回路Bの機能（動作）を記述することはできるが，記述した機能Aと機能Bの処理は記述した順番に処理されてしまう．そのため，プログラミング言語では並列動作する回路の処理を正確に記述することができない．

〔図2〕無限ループは記述できない（してはいけない）

挙げられます．

その結果，いままでは個別に設計していた，プロセッサ，メモリ，アナログ回路，専用回路（application specific circuits：ASIC）などの部品を一つのチップとして設計するようになってきました．また，設計期間を短縮するために，設計するチップ上で実行されるソフトウェアも同時に開発されるようになってきました．つまり，このことは，「システム全体」を設計することを意味しています．現状では，性能とコスト，消費電力，サイズなどの制約条件をすべて満たすためには，システム全体を1チップ化することが不可欠となっています．このようにシステム全体を1チップに実装することをSoC（System on Chip）と呼びます．

システムの設計は，プロセッサやASICなどの設計に比べて，はるかに規模が大きくなります．また，システムの機能のうち，どの機能をハードウェアで実現するか，どの機能をソフトウェアで実現するかを決める問題もあります．そのため，これまでの設計に比べて，システム設計は複雑で，設計期間も長くなってしまいます．そこで，システムを効率良く設計する手法の開発が求められるようになってきました．この要求に対する一つの答が，SystemCなのです．

● SystemCとプログラミング言語の違い

上記でも述べましたが，SystemCは，C++言語をベースにしたシステムレベル設計言語です．このSystemCとプログラミング言語であるC++言語との違いはどこにあるのでしょうか？　また，ハードウェアの設計に用いられているVerilog-HDLやVHDLといったハードウェア記述言語（Hardware Discription Language：HDL）とはどこが違うのでしょうか？そこで，まずSystemCとプログラミング言語との違いについてみていきましょう．

▶並列動作の記述

C++言語やC言語に限らずほとんどのプログラミング言語では，記述した処理が逐次的に処理されていきます．一方ハードウェアでは，逐次的に動作する部分もありますが，ほとんどの処理は並列的に動作します．

図1に示すような並列に動作する回路がシステム内にあるとします．これをプログラミング言語で記述した場合，**図1**に示すように回路Aと回路Bの動作は，それぞれ機能Aと機能Bとして記述することができます．しかし，プログラミング言語では記述した順番に処理されていくため，機能Aと機能Bは記述した順番で処理されることになります．したがって，回路Aと回路Bの動作を正確に記述することができません．SystemCではHDLと同様に，並列動作を記述するための構文が用意されていて，簡単に並列動作を記述することができます．

▶データ型

ソフトウェアでは，基本的に整数型や実数型のデータ型を中心に扱いますが，ハードウェアでの処理はビット型が中心になります．SystemCでは，ビット型とビットベクタ型をもっており，これらに対してビット連結，特定ビットの指定などさまざまな処理を行うことができます．また，ビット型やビットベクタ型を用いることによって，回路の入出力，回路内の信号線のビット幅を細かく指定することができます．

▶信号への同期

（システム内の）ハードウェアの多くは，クロック信号や制御信号に同期しています．ここでの同期とは，信号の値が変化したときに，動作を開始することを意味しています．プログラミング言語では，このような信号（変数）に同期した動作を記述しようとするとプログラムが複雑になってしまいます．一方，SystemCでは，信号（変数）の変化に反応する構文が用意されていますから，クロック信号や制御信号への同期を簡潔に記述することができます．

● SystemCとハードウェア記述言語（HDL）の違い

続いて，SystemCとHDLとの違いを見ていきます．

▶記述の制限

HDLは基本的に，ハードウェアを設計することを目的とした言語です．ですから，その記述は実際のハードウェアを表現していることになります．そのため，HDLの記述には多くの制限があります．

たとえば，**図2**に示すように無限ループを記述した場合，無限ループ内に記述したハードウェアが無数に存在することを意

味します．したがって，文法上の問題はなくても，無限ループを記述することはできません．これと同じ理由で，何回ループするかわからないwhile文なども記述することはできません．SystemCでは，ソフトウェアの記述に関しては制限はありません．ただし，ハードウェアを記述する際には，上記の点について注意する必要があります．

▶**シミュレーション時間**

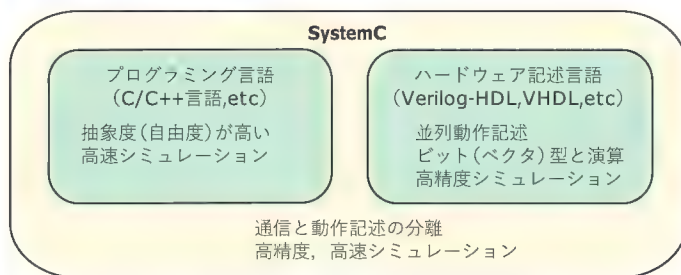
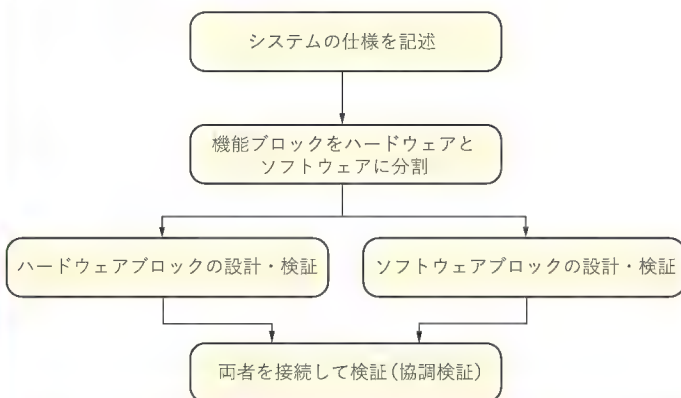
HDLで記述したハードウェアをシミュレーションする際には，記述したハードウェアの動作を正確にシミュレーションします．そのため，回路の入力から出力に至るまでに通過する信号線やゲート回路[注1]の遅延時間を計算することになります．その結果，シミュレーションに非常に時間がかかってしまうことになります．とりあえず回路が正しく動作するかどうかだけを確認したい場合には，回路の動作時間などの情報は必要ありません．SystemCでは，そのような場合，プログラミング言語(C++言語)に近い記述で回路の動作を記述します．この場合，単純に信号の変化だけがシミュレートされるので，高速なシミュレーションを行うことができます．

〔図3〕**SystemCの言語機能**

SystemC

プログラミング言語
(C/C++言語,etc)
抽象度(自由度)が高い
高速シミュレーション

ハードウェア記述言語
(Verilog-HDL,VHDL,etc)
並列動作記述
ビット(ベクタ)型と演算
高精度シミュレーション

通信と動作記述の分離
高精度，高速シミュレーション

〔図4〕**システム設計の設計フロー**

従来のシステム設計では，ハードウェアブロックとソフトウェアブロックの設計には異なる言語を用いて設計するため，システム仕様の変更には，多くの作業を必要とする．また，ソフトウェアブロックの設計・検証は，ハードウェアブロックの設計がある程度完了していないと行うことができないため，実際には，「ハードウェアブロック設計・検証→ソフトウェアブロック設計・検証→協調検証」の順に設計が進められる．

● SystemCの特徴

またSystemCには，システムの設計を効率良く行うための機能が盛り込まれています．続いて，SystemCの特徴的な機能について見ていきましょう．

▶**通信と動作の分離**

システムの内部は，いくつかの機能ブロックに分けることができます．システムを設計する際には，分割した機能ブロックを個別に設計することになります．そして，機能ブロック間は何らかの方法でデータをやりとりする必要があります．このデータの送受信の方法を，各機能ブロック内で実装した場合には，データの通信方法が変更されるたびに，それぞれの機能ブロックの設計をやり直す必要があります．

これを防ぐためSystemCでは，チャネル，インターフェース，ポートという概念を導入して，機能ブロックの動作と機能ブロック間の通信部分を完全に切り離して記述します．これによって，機能ブロックの動作と機能ブロック間の通信部分のそれぞれを独立に修正することができます．

▶**高精度，高速のシミュレーション**

先ほども解説しましたが，一般に，実際のハードウェアに近い記述をすると，回路内部の細かい部品の動作まで正確にシミュレーションできるため，精度の高いシミュレーション結果を得ることができます．しかし，その反面シミュレーションに非常に時間がかかってしまいます．一方，動作のみのソフトウェアに近い記述をすると，信号(変数)の変化だけをシミュレーションするため，高速にシミュレーションを行うことができます．しかしこの場合には，精度の高いシミュレーション結果を得ることができません．

SystemCでは，ソフトウェアとハードウェアの両方を記述することができます．この特徴を生かして，システム全体をシミュレーションする際に，精度の高いシミュレーションを行いたい機能ブロックだけをハードウェアに近い記述にし，それ以外の機能ブロックをソフトウェアに近い記述にします．これによって，高精度かつ高速なシミュレーションを実現することができます．

以上の説明から，SystemCは**図3**に示すように，プログラミング言語とHDLの両方の特徴を併せもっていることがわかります．

2 SystemCによるシステム設計

SystemCは，システム上のすべての機能を記述することができます．システムを一つの言語だけで記述できるということは，システムを設計する際に，大きな利点となります．一般にシステム設計は，**図4**に示すような設計フローに基づいて行われていきます．

注1：ゲート回路については，第2章を参照のこと．

● 従来のシステム設計

プログラミング言語とHDLを用いてシステムを設計する従来のシステム設計の場合，まず，システムの仕様をUML (Unified Modeling Language) などのシステム仕様記述言語や自然言語で記述します．システムの仕様が確定したら，システムの各機能ブロックをハードウェアブロックとソフトウェアブロックに分割します．通常，システム設計者がコストや消費電力などの制約条件を考慮して分割を行います．

分割後は，ハードウェアブロックはHDL，ソフトウェアブロックはプログラミング言語を用いて設計していきます．ハードウェアブロックとソフトウェアブロックを異なる言語を用いて設計するため，ハードウェア(ソフトウェア)ブロックの設計仕様の変更が，ソフトウェア(ハードウェア)ブロックの設計仕様に影響を与える場合，ソフトウェア(ハードウェア)ブロックのどの部分に影響を及ぼすのか見当がつきにくくなります．その結果，設計仕様の変更やシステムの再分割には多くの手間が必要になります．

それぞれの設計が完了したら，ハードウェアブロックとソフトウェアブロックの設計を接続して協調検証を行います．ここでも，ハードウェアとソフトウェアのそれぞれの開発環境から協調検証のための環境に移植する手間が生じてしまいます．

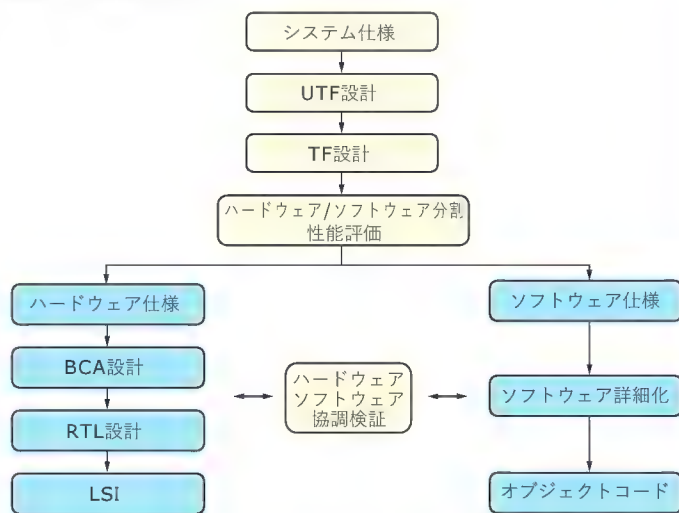
また，**図4**の設計フローでは，ハードウェアブロックの設計・検証とソフトウェアブロックの設計・検証は並行して進められています．しかし実際には，ソフトウェアブロックの設計はハードウェアブロックの設計がある程度完了していないと行えないため，ハードウェアブロックの設計・検証の後，ソフトウェアブロックの設計・検証が行われることになります．

● SystemCを用いたシステム設計

次に，SystemCを用いた場合のシステム設計の設計フローを**図5**に示します．SystemCによるシステム設計では，システムの仕様からハードウェアブロック，ソフトウェアブロックの設計まで，すべて同じ言語・開発環境で行うことができます．このため，異なる開発環境からの移植作業などの手間が省けるので，従来の設計手法に比べて，はるかに効率良くシステム設計を行うことができます．

大規模かつ複雑なシステム設計では，まず，大まかな機能と構造を決めておき，設計を進めていきながら実際の回路に近づけていくほうが効率良く設計することができます．なお，大まかな機能(動作)と構造だけが決まっている設計(記述)は抽象度が高い設計(記述)，細かい機能(動作)や構造が決まっている設計は抽象度が低い設計(記述)といいます．先ほど述べたように，抽象度の高い設計から抽象度の低い設計に詳細化していく設計手法のことをトップダウン設計(top-down design)手法と呼びます．

SystemCを用いたシステム設計でも，トップダウン設計手法によって設計が進められていきます．このトップダウン設計手法を円滑に進めるために，SystemCでは，**図5**に示す四つの異

〔図5〕SystemCを用いたシステム設計の設計フロー

システム設計のすべての行程をSystemCだけで行うことができる．SystemCを用いたシステム設計では，抽象度の高い設計モデルからステップバイステップで抽象度の低い設計モデルに詳細化していく，トップダウンの設計方式となる．協調検証の際には，設計の抽象度に応じたシミュレーションを行うことによって，効率的に検証作業を進めることができる．

なる抽象度の設計モデルをサポートしています．SystemCでサポートしている四つの設計モデルは，次のようになります．

▶ **UTF (Untimed Functional) モデル**

UTFモデルは，もっとも抽象度が高い設計モデルです．UTFモデルでは，システム内の各機能ブロックがどのような機能を実現しているかが記述されているだけで，各機能ブロックがどの順番で動作するかは記述されていません．つまり，時間の概念を含まない記述となります．

▶ **TF (Timed Functional) モデル**

TFモデルは，UTFモデルに次いで抽象度が高い設計モデルです．TFモデルは，各機能ブロックが動作する順番が記述されています．UTFモデルに時間の概念を付け加えた設計モデルと考えればよいでしょう．

▶ **BCA (Bus Cycle Accurate) モデル**

BCAモデルは，3番目に抽象度が高い設計モデルです．BCAモデルでは，各機能ブロック間を接続するバスに関して，クロックに対して正確に動作するように記述されています．すなわち，システムの機能ブロック間のデータの流れを正確に記述した設計モデルとなります．

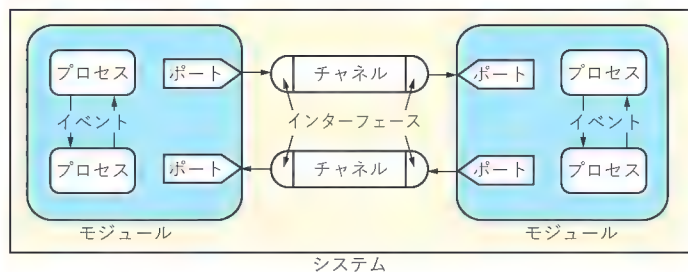
▶ **RTL (Register Transfer Level) モデル**

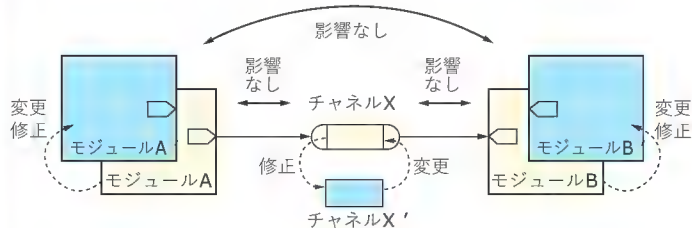
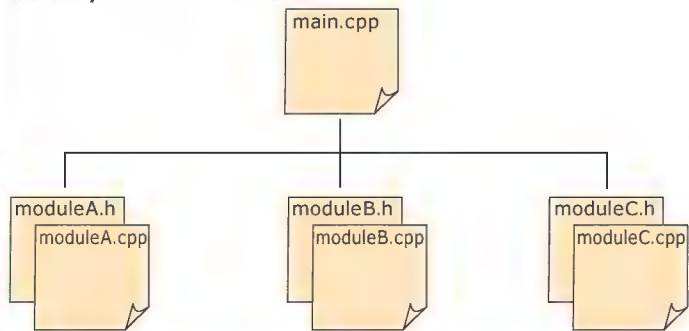
RTLモデルは，もっとも抽象度の低い設計モデルです．RTLモデルでは，機能ブロック内の動作についても，クロックに対して正確に動作するように記述されています．HDLを用いてハードウェアを記述した場合と，ほぼ同じ抽象度になります．

なお上記の設計モデルのうち，BCAモデルとRTLモデルに

関しては，ハードウェアに関する抽象度となっています．ソフトウェアに関する抽象度については，SystemC 3.x から対応される予定となっています．

SystemC を用いたシステム設計では，**図5**に示すように，まず，システム仕様に基づいて，システムの機能をブロック化したUTFモデルを設計します．続いて，各機能ブロックに対して順序付けを行い，時間の概念を追加したTFモデルを設計していきます．そして，この段階で，システムをハードウェアブロックとソフトウェアブロックに分割します．その後，ハードウェアブロックの設計に関しては，BCAモデルからRTLモデルへとより抽象度の低い，すなわち，より精度の高い設計に変更していきます．このBCAモデルとRTLモデルの設計の際には，ソフトウェアブロックとの協調検証を行い，設計の変更・修正を行っていきます．最後に，ハードウェアブロックはLSI化され，ソフトウェアブロックはオブジェクトコードにコンパイルされることになります．

〔図6〕SystemC でのシステムの表現

SystemC では，システムの各機能ブロックをモジュール，モジュール間の通信部分をチャネルとして表現する．また，上位のモジュールから下位のモジュールを呼び出す（インスタンス化する）ことによって，階層構造を表現できる．モジュールとチャネルは，ポートとインターフェースを介して接続される．モジュール内のプロセスは，モジュールの動作を表し，プロセス間はイベントによって通信が行われる．

〔図7〕モジュール（動作）とチャネル（通信）の分離

〔図8〕SystemC のファイル構成

SystemC では，モジュールごとにヘッダファイルとインプリメンテーションファイルを用意する．ヘッダファイルには，モジュールやモジュールのポートとプロセスなどを記述する．インプリメンテーションファイルには，モジュールのプロセスの動作を記述する．ファイルの最上位には，モジュールの接続関係や接続に用いるチャネル，シミュレーション命令などを記述するシステムファイルを用意する．

3 SystemC によるシステムの記述

● SystemC によるシステムの表現

SystemC では，いったいどのようにしてシステム全体を表現しているのでしょうか？　ここでは，SystemC によるシステムの表現方法について説明します．

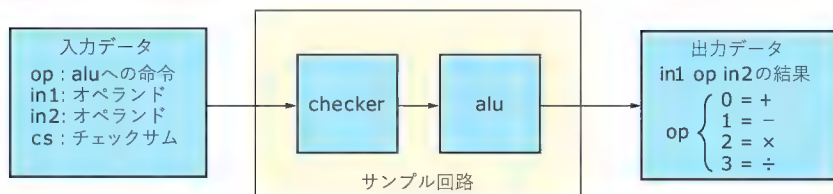
SystemC では，**図6**のように，いくつかのモジュールとそれらを接続するチャネルでシステム全体を表現しています．モジュールはシステムの機能ブロックを表し，チャネルはモジュール間を接続する通信部分を表しています．システムをいくつかのモジュールで表現することによって，複雑なシステムを理解しやすくしています．また，モジュール内で下位モジュールを呼び出すことで階層構造が表現できます．

各モジュールには，いくつかのプロセスが含まれています．プロセスは，モジュールの機能や動作の基本単位を表し，それぞれのプロセスは並列に動作します．また，プロセス同士はイベントを使って通信を行います．このプロセスを用いることによって，ハードウェアの並列処理を表現できます．

なおモジュールとチャネルは，**図6**に示すようにポートとインターフェースを介して接続しています．これは，機能（モジュール）と通信（チャネル）の記述を分離するために，このような構成になっています．この構成により，それぞれのモジュールとチャネルは他のモジュールやチャネルから，内部のデータやアルゴリズムを隠蔽することができます．また**図7**に示すように，モジュールやチャネルの変更が，他のモジュールとチャネルに影響を及ぼさないようになっています．このため，各モジュール（チャネル）の設計者は，他のモジュールやチャネルの構造を気にすることなく，モジュール（チャネル）の設計を行うことができます．

● SystemC のファイル構成

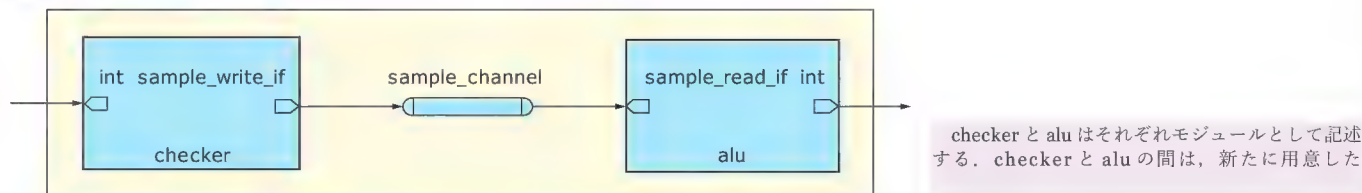
続いて，SystemC のファイル構成を**図8**に示します．SystemC では，システム内の各モジュールをヘッダファイルとインプリメンテーションファイルの二つのファイルで記述します．ヘッダファイルには，モジュールやモジュールのポートとプロセスなどを記述します．一方，インプリメンテーションファイルには，モジュールのプロセスの動作を記述することになります．すなわち，ヘッダファイルには回路の入出力と機能，インプリ

〔図9〕サンプル回路の仕様

サンプル回路は，二つの機能ブロック checker と alu をもつ．サンプル回路には，クロックごとに整数データが入力される．1クロック目(op)の整数データは alu への演算命令，2クロック目(in1)と3クロック目(in2)の整数データは演算命令のオペランドデータ，4クロック目(cs)の整数データは前の三つの整数データに対するチェックサムとなっている．以降，これらの整数データが繰り返し入力される．checker は，入力データのチェックサムを計算し，入力データに誤りがないかを調べる．alu は，checker から受け取った三つの整数データに対して**表1**をもとに演算を行い，その演算結果を出力する．

〔表1〕サンプル回路の演算命令

op	実行される演算
0	in1 + in2
1	in1 − in2
2	in1 × in2
3	in1 ÷ in2

〔図10〕SystemC でのサンプル回路

checker と alu はそれぞれモジュールとして記述する．checker と alu の間は，新たに用意した

メンテーションファイルには回路の動作を記述すると考えればよいでしょう．

なお，ファイル構成の最上位には，システム全体の構成を記述するシステムファイルが必要となります．システムファイルには，モジュール間の接続関係やモジュールの接続に用いるチャネルなどを記述します．このほか，記述したシステムに対するシミュレーションの実行命令なども記述します．

4 SystemCによる設計の例

ここでは，**図9**のサンプル回路の記述例を示し，SystemC の基本的な文法について見ていきます．

● サンプル回路の概要

今回設計するサンプル回路は，入力データに対する簡単な誤り検出機構が付いた算術演算回路(arithmetic and logic unit：ALU)です．サンプル回路は，入力と出力を一つずつもち，このほかにクロックが入力されます．この内部には，checker と alu の二つの機能ブロックをもちます．

サンプル回路には，クロックごとに一つの整数データが入力されます．1クロック目の整数データは，機能ブロック alu に対する演算命令，2クロック目と3クロック目の整数データは，1クロック目の整数データで指定された演算命令のオペランドとなっています．4クロック目の整数データは，前の三つの整数データに対するチェックサム(check sum)になっています．5クロック目以降は，四つの整数データが繰り返し入力されることになります．なお，指定できる演算命令を**表1**に示します．表中の op は1クロック目の整数データの値，in1 と in2 はそれぞれ，2，3クロック目の整数データを表しています．また，サンプル回路のチェックサムは単純に三つの整数データの総和としています．

機能ブロック checker は，入力された整数データの総和を求め，チェックサムを計算します．4クロック目に入力されたチェックサムと計算したチェックサムが異なっていれば，それまでに入力された整数データを破棄します．入力されたチェックサムが正しければ，三つの整数データは機能ブロック alu に入力されます．そして alu は，checker から受け取った三つの整数データに対して**表1**にしたがった演算を行い，その演算結果を外部に出力します．

● サンプル回路の記述

サンプル回路を SystemC で記述すると，**図10**に示すようなイメージになります．サンプル回路の二つの機能ブロックをモジュールとして記述し，モジュール間はチャネルによって接続します．ここでモジュール間を接続するチャネルは，新たに用意した sample_channel チャネルを使用することにします．

● モジュールのヘッダファイルの記述

まず，モジュール checker とモジュール alu の記述を例に，モジュールの記述方法について解説していきます．モジュール checker とモジュール alu のヘッダファイルとインプリメンテーションファイルを**リスト1～リスト4**にそれぞれ示します．

▶**モジュール定義**

モジュールの定義は，ヘッダファイルに次の形式で記述します．

```
SC_MODULE(モジュール名){
  //ポート宣言とチャネル接続
  //プロセスの宣言
  //イベントの宣言
  //メンバ変数の宣言
  ...
  SC_CTOR(モジュール名){
    //コンストラクタ
    //モジュールタイプの指定
    //プロセスのセンシティビティリスト
    ...
  }
}
```

なお，SC_CTOR()はモジュールのコンストラクタを表しています．コンストラクタは，モジュールがインスタンス化されるときに呼び出される特別な関数です[注2]．後で詳しく説明しますが，ここにプロセスのプロセスタイプとセンシティビティリストを記述します．

▶ポート宣言とチャネル接続

続いて，モジュール定義の中に，ポート宣言とチャネル接続

注2：コンストラクタの詳細については，C++言語の参考書などを参照いただきたい．

〔リスト1〕checkerのヘッダファイルの記述例（checker.h）

```
SC_MODULE(checker){

  sc_in_clk clk;
  sc_in<int> in;
  sc_port<sample_write_if<int> > out;

  void check_sum_checker(void);

  int data_num;
  int check_sum;

  SC_CTOR(checker){
    SC_THREAD(check_sum_checker);
    sensitive << clk.pos();
  }
};
```

〔リスト2〕checkerのインプリメンテーションファイルの記述例（checker.cpp）

```
#include "systemc.h"
#include "sample_channel_if.h"
#include "sample_channel.h"
#include "checker.h"

void checker::check_sum_checker()
{
  int i;
  while(1){
    wait();
    i = in.read();
    if (data_num < 3) {
      check_sum += i;
      out->s_write(i);
      data_num++;
    } else if (data_num == 3) {
      if (check_sum != i) {
        out->s_reset();
      }else{
        out->s_write(i);
      }
      data_num = 0;
      check_sum = 0;
    }
```

〔リスト3〕aluのヘッダファイルの記述例（alu.h）

```
SC_MODULE(alu){

  sc_in_clk clk;
  sc_port<sample_read_if<int> > in;
  sc_out<int> out;

  void fetch_decode(void);
  void adder(void);
  void subtracter(void);
  void multiplyer(void);
  void divider(void);

  sc_event add,sub,mult,div;

  int opcode[4];

  SC_CTOR(alu){
    SC_THREAD(fetch_decode);
    sensitive << clk.pos();
    SC_THREAD(adder);
    SC_THREAD(subtracter);
    SC_THREAD(multiplyer);
    SC_THREAD(divider);
  }
};
```

〔リスト4〕aluのインプリメンテーションファイルの記述例（alu.cpp）

```
#include "systemc.h"
#include "sample_channel_if.h"
#include "sample_channel.h"
#include "alu.h"

void alu::fetch_decode()
{
  int i;
  while (1) {
    wait();
    if(in->rest_data_num() == 4){
      for (i=0; i<4; i++) {
        in->s_read(opcode[i]);
      }

      switch(opcode[0]){
        case 0 : add.notify();  break;
        case 1 : sub.notify();  break;
        case 2 : mult.notify(); break;
        case 3 : div.notify();  break;
      }
    }
  }
}

void alu::adder()
{
  int i;
  while (1) {
    wait(add);
    i = opcode[1] + opcode[2];
    out.write(i);
  }
}

void alu::subtracter()
{
  int i;
  while (1) {
    wait(sub);
    i = opcode[1] - opcode[2];
    out.write(i);
  }
}

void alu::multiplyer()
{
  int i;
  while (1) {
    wait(mult);
    i = opcode[1] * opcode[2];
    out.write(i);
  }
}

void alu::divider()
{
  int i;
  while (1) {
    wait(div);
    i = opcode[1] / opcode[2];
    out.write(i);
  }
}
```

を記述していきます．このポートは，モジュールの入出力を表現しています．そして，モジュールがどのチャネルを使って他のモジュールと接続するかを，このポート宣言で定義します．ポート宣言の記述は，以下のようになります．

```
//入力ポート
sc_in<データ型> ポートインスタンス名;
//出力ポート
sc_out<データ型> ポートインスタンス名;
//クロックのための特別な記述
sc_in_clk clk;
//チャネルへの接続
sc_port<チャネルインターフェース名>
                              ポートインスタンス名;
```

モジュールcheckerの入出力には，クロック入力，整数データの入力，モジュールaluへの出力があります．これらの入出力のポートが**リスト1**の先頭部分に宣言されています．モジュールcheckerの出力はsample_channelチャネルに接続されるため，sc_outではなくsc_portで出力ポートを宣言します．チャネルインターフェース名に指定しているsample_write_if<int>については，後ほど説明します．

```
sc_port<sample_write_if<int> > out;
                            └空白入れる
```

なお上記のように，ポート宣言やチャネル接続を記述する際に“>”が続く場合，必ず間に空白を挿入しておく必要があります．この空白を挿入しておかないとコンパイル時にエラーとなってしまいます．

▶プロセス宣言

ポートの宣言の次には，プロセスの宣言を記述します．プロセスの宣言は，以下のようにして記述します．なお，プロセスの引き数と戻り値は必ずvoidとしておきます．

```
void プロセス名(void);
```

モジュールcheckerでは一つのプロセスを宣言していますが，その引き数と戻り値がvoidとなっていることがわかります．

またプロセスは，モジュールのコンストラクタ(SC_CTOR)内にタイプの宣言とセンシティビティリストを記述する必要があります．プロセスがモジュールのコンストラクタ内で呼び出されている理由は，プロセスがモジュールの動作の基本単位であるため，そのモジュールが呼び出されるときは必ずプロセスが実行されなければならないからです．

また，センシティビティリストは，直前に記述したプロセスを開始させる信号線の一覧を表しています．センシティビティリストに記述された信号線の値が変化すると，直前に記述したプロセスの動作が開始されます．つまり，プロセスの動作を引き起こすトリガとなる信号線がセンシティビティリストに記述されることになります．プロセスタイプの宣言とセンシティビティリストは，次のようにして記述します．

```
プロセスタイプ(プロセス名);
//信号線の変化に反応
sensitive << 信号線名 (<< これで追加可能);
```

プロセスのタイプには，Threadプロセス(SC_THREAD)，Methodプロセス(SC_METHOD)，Clocked Threadプロセス(SC_CTHREAD)の3種類があり，それぞれ異なった動作をします．

このうちThreadプロセスとMethodプロセスは，基本的にセンシティビティリストに記述した信号線に同期して動作します．Threadプロセスでは，wait文でプロセスの動作を中断させることができ，Methodプロセスではwait文を記述できないという違いがあります．これらのプロセスに関しては，後ほど詳しく説明します．

Clocked Threadプロセスは，クロックにのみ同期するプロセスです．このClocked Threadプロセスは，旧バージョンであるSystemC 1.0との互換性を保つために用意されたプロセスなので，通常は使わないほうがよいでしょう．

モジュールcheckerでは，プロセスタイプをThreadプロセスとして宣言しています．また，センシティビティリストにclk.pos()と記述していますが，この記述はクロック(clk)の立ち上がり(信号の値が0から1に変化する瞬間)にプロセスが動作することを意味しています．なお，クロックの立ち下がり(信号の値が1から0に変化する瞬間)に動作させたい場合は，clk.neg()と記述します．

▶イベントとメンバ変数の宣言

続いて，イベントの宣言を記述していきます．イベントの宣言は次のように記述します．なお，イベントはプロセス間同士の通信にのみ使用でき，モジュール間の通信には使用することはできません．

```
sc_event イベント名;
```

モジュールcheckerには，プロセスが一つしかないため，イベントを宣言する必要がありません．モジュールaluでは，実行する演算命令を決定するfetch_decode()プロセスと演算を実行する四つのプロセスとの間で通信を行うため，四つのイベントが宣言されています．

メンバ変数の宣言は，C++言語と同様に記述することができます．モジュールcheckerでは，チェックサムの途中の計算結果を保持するためのメンバ変数と，入力されたデータ数をカウントするためのメンバ変数が宣言されています．

以上でモジュールのヘッダファイルの記述が完了します．

● モジュールのインプリメンテーションファイルの記述

ヘッダファイルの記述が終了したら，プロセスの動作をインプリメンテーションファイルに記述していきます．

▶ヘッダファイルのインクルード

インプリメンテーションファイルの先頭部分に，SystemCのライブラリとヘッダファイルをインクルードします．なおSystemCのライブラリは，ヘッダファイルの先頭部分でインク

ルードしてもかまいません．

```
#include "systemc.h"
#include "ヘッダファイル名"
```

モジュール checker では，新たに用意した sample_channel チャネルを使用しているため，sample_channel チャネルとそのインターフェースを記述しているヘッダファイルもインクルードする必要があります．これらの記述については後ほど説明します．

▶**プロセスの動作記述**

ライブラリとヘッダファイルのインクルードの記述に続いて，ヘッダファイルで宣言した各プロセスの動作を記述していきます．各プロセスの動作は，モジュールのメンバ関数として定義します．

ここでプロセスの動作記述について詳しく説明します．プロセスの動作記述は，そのプロセスタイプによって大きく異なります．

Thread プロセスは，一度最後まで実行した後は，再び呼び出されることがありません．そのため，動作全体を無限ループの中に記述する必要があります．while 文内の先頭に wait() 文を挿入することで，プロセスが動作するときはいつでも，最初の動作から実行されることになります．したがって，Thread プロセスの記述は以下のようになります．

```
void モジュール名::プロセス名（void）{
  while(1){
    wait();
    //プロセスの動作
  }
}
```

ここで wait 文の () 内に，イベント名を記述するとそのイベントに同期して動作するようになります．この記述は，先に説明したセンシティビティリストと同じような働きをすることになります．このように，wait 文の () 内にイベントを記述する方法を動的センシティビティと呼びます．これに対して，ヘッダファイル内に記述するセンシティビティを，静的センシティビティと呼びます．

Method プロセスの場合は，センシティビティリストの信号線が変化するたびに呼び出されるので，無限ループを記述する必要はありません．また，Method プロセスは呼び出されると，必ず記述したすべての動作を実行するため wait 文を記述してはいけません．そこで処理が止まってしまい，シミュレーションができなくなるからです．Method プロセスの記述は以下の形式となります．

```
void モジュール名::プロセス名(void){
  //プロセスの動作
  //wait 文は記述しないこと！
}
```

なおプロセスタイプは，インプリメンテーションファイル中には明記されません．そのため，プロセスの動作を記述するときは，そのプロセスタイプに十分に気を付けて記述する必要があります．

モジュール checker のプロセス check_sum_checker() は Thread プロセスとして，ヘッダファイルで宣言されています．プロセス check_sum_checker() は，静的センシティビティリストに clk.pos() と記述されていますから，クロックの立ち上がりのたびに，while 文内の動作が実行されることになります．

▶**イベントによるプロセス間通信**

一方，モジュール alu のプロセス adder() では，

```
wait(add);
```

のように，動的センシティビティが記述されています．このことから，このプロセス adder() は，イベント add に反応して動作することがわかります．実際に，プロセス adder() は，プロセス fetch_decode() 内の

```
case 0 : add.notify(); break;
```

が実行されたときに動作することになります．以下に示すように，

```
イベント名.notify();
```

と記述することで，そのイベントが発生したことを他のプロセスに通知することになります．これによってプロセス間の通信を行います．

▶**入出力の読み書き**

プロセスでは，モジュールの入力に対して何らかの処理を行い，その結果を出力します．このとき，モジュールの入出力ポートに対して，データの読み書きを行う必要があります．入出力ポートへの値の読み書きは以下のように記述します．

```
入力ポート名.read()          //値の読み込み
出力ポート名.write(出力値)   //値の書き込み
```

同じデータ型の入出力ポート同士の単純な代入を行う場合には，このような記述をする必要はありません．

しかし，異なるデータ型の信号線への代入や制御文の条件式内では，上記の記述を省略することができません．記述を省略した場合には，コンパイル時にエラーとなってしまいます．このような問題を避けるためにも，必ず，上記の記述で値を読み書きするように心がけてください．なお，メンバ変数やプロセス関数内で宣言した変数に対しては，上記のように記述する必要はありません．

このほか，プロセスでは，入出力チャネルに対するデータの読み書きも行う必要があります．入出力チャネルに対してのデータの読み書きは以下のように記述します．

```
チャネルポート名->通信関数名(引き数)
```

チャネルへのデータの読み書きは，チャネルのインターフェース定義の中で宣言された通信関数を用いて行います．詳細については後のチャネルの記述例で解説します．

以上がモジュールのインプリメンテーションファイルの記述

〔リスト 5〕sample_channel インターフェースの記述例（sample_channel_if.h）

```
template <class T=int>
class sample_read_if : public sc_interface {
public:
  virtual void s_read(T &) = 0;
  virtual int rest_data_num() = 0;
};

template <class T=int>
class sample_write_if : public sc_interface {
public:
  virtual void s_write(T) = 0;
  virtual void s_reset() = 0;
};
```

になります．

● チャネルの記述

今回のサンプル回路では，モジュール間を接続するチャネルを新たに用意しました．ここでは新たに用意した sample_channel チャネルの記述を例に，チャネルの記述方法について解説していきます．なお，sample_channel チャネルのインターフェースの記述と sample_channel チャネルの記述を**リスト 5** と**リスト 6** に示します．

▶インターフェースの定義

チャネルを使用するためには，モジュールとチャネルを接続するインターフェースを定義する必要があります．インターフェースは，SystemC に用意されている sc_interface クラスを継承します．この継承とは，既存のクラスに新たな機能を加えた新たなクラスを定義することを意味します注3．すなわちこの場合，SystemC に用意されている（基本的なインターフェースである）sc_interface に機能を追加した新たなインターフェースを定義することを意味します．インターフェースの定義は，以下のように記述します．

```
class インターフェース名 : public sc_interface {
public:
  //通信関数の宣言
  virtual 戻り型 通信関数名(引き数型) = 0;
};
```

なお，通信関数は virtual 関数として宣言します．また，宣言する通信関数は複数あってもかまいません．定義するインターフェースの数も自由です．今回の例では，読み込み用と書き込み用の 2 種類のインターフェースを用意しましたが，これらのインターフェースを一つにまとめることも可能です．このほか，例に示すように，インターフェースのテンプレートを記述して，いろいろなデータ型に対応させて汎用性を高めることもできます．このインターフェースや後述するチャネルの記述に関しては，C++ 言語の知識がかなり要求される部分となっています．

▶チャネルの定義

インターフェースの記述が終了したら，チャネルの定義を行います．チャネルもインターフェースと同様に SystemC に用意されている sc_channel クラスを継承します．また，先ほど定義したインターフェースも継承します．したがって，チャネルは基本的に多重継承を使用することになります．チャネルの記述は以下のようになります．

```
class チャネル名 : public sc_channel,
    public インターフェース名 ...
{
public:
  //チャネルコンストラクタ
  //通信関数の定義
private:
  //イベント，メンバ変数の宣言
};
```

〔リスト 6〕sample_channel の記述例（sample_channel.h）

```
template <class T=int>
class sample_channel : public sc_channel,
    public sample_read_if<T>,
    public sample_write_if<T>
{
public:

  sample_channel(char* channame = "none")
    : sc_channel(channame){
    data_num = 0;
    top = 0;
  }

  void s_read(T& i){
    if (data_num == 0) {
      wait(write_event);
    }
    i = data[top];
    data_num--;
    top = (top+1) % max;
    read_event.notify();
  };

  void s_write (T i){
    if (data_num == max) {
      wait(read_event);
    }
    data[(top + data_num) % max] = i;
    data_num++;
    write_event.notify();
  };

  int rest_data_num(){
    return data_num;
  };

  void s_reset(){
    top = 0;
    data_num = 0;
  };

private:
  enum e {max = 10};
  T data[max];
  int data_num;
  int top;
  sc_event read_event,write_event;
};
```

注 3：コンストラクタと同様に，継承の詳細については C++ 言語の参考書などを参照すること．

〔リスト7〕サンプル回路のシステムファイルの記述例(main.cpp)

```
#include "systemc.h"
#include "sample_channel_if.h"
#include "sample_channel.h"
#include "checker.h"
#include "alu.h"

int sc_main(int argc, char *argv[])
{

  int i;
  sc_clock clk("clk", 100, SC_NS, 0.5,0, SC_NS, false);
  sc_signal<int> in;
  sc_signal<int> out;
  sample_channel<int> chk_to_alu;

  checker CHECKER("checker");
  CHECKER.clk(clk);
  CHECKER.in(in);
  CHECKER.out(chk_to_alu);

  alu ALU("alu");
  ALU.clk(clk);
  ALU.in(chk_to_alu);
  ALU.out(out);

  sc_trace_file *trace_f;
  trace_f = sc_create_vcd_trace_file("sample_trace");
  ((vcd_trace_file *)trace_f)->sc_set_vcd_time_unit(-9);
  sc_trace(trace_f,clk,"clk");
  sc_trace(trace_f,in,"in");
  sc_trace(trace_f,out,"out");

  for (i=1; i<30; i++) {
    switch (i) {
      case 1  : in = 0;  break;
      case 2  : in = 2;  break;
      case 3  : in = 1;  break;
      case 4  : in = 3;  break;

      case 5  : in = 1;  break;
      case 6  : in = 2;  break;
      case 7  : in = 10; break;
      case 8  : in = 13; break;

      case 9  : in = 2;  break;
      case 10 : in = 12; break;
      case 11 : in = 12; break;
      case 12 : in = 26; break;

      case 13 : in = 3;  break;
      case 14 : in = 55; break;
      case 15 : in = 5;  break;
      case 16 : in = 63; break;

      case 17 : in = 5;  break;
      case 18 : in = 4;  break;
      case 19 : in = 3;  break;
      case 20 : in = 12; break;

      case 21 : in = 2;  break;
      case 22 : in = 4;  break;
      case 23 : in = 3;  break;
      case 24 : in = 12; break;

      case 25 : in = 2;    break;
      case 26 : in = 1024; break;
      case 27 : in = 99;   break;
      case 28 : in = 1125; break;
      default : in = 0;  break;
    }
    sc_start(100,SC_NS);
    cout << i << ": IN  DATA = " << in << endl;
    cout << i << ": OUT DATA = " << out << endl;
  }
  return 0;
}
```

なお，チャネル内の通信関数同士は，プロセスと同様に，イベントを使って通信を行うことができます．

sample_channelチャネルは，単純なバッファの役割を果たします．sample_write_ifインターフェースの通信関数s_write()によってデータが書き込まれ，sample_read_ifインターフェースの通信関数s_read()によって書き込まれたデータを読み出します．s_write()の定義では，データを書き込む際にバッファの容量が満杯のときには，バッファからデータが読み出されるまで書き込みを中断します．バッファの読み出しがあったかどうかは，read_eventイベントによって通知されます．そして，read_eventイベントが発生したら，データの書き込みを再開します．同様にs_read()の定義では，バッファが空のときには，write_eventイベントが発生するまで，バッファの読み込みを中断するようにしています．この他，バッファを初期化するs_reset()とバッファ内のデータの個数を知らせるrest_data_num()を用意しています．

今回は説明のために，sample_channelチャネルを用意しましたが，SystemCの基本チャネルには，このsample_channelチャネルと同じような機能をもつsc_fifoチャネルが用意されています．

● システムファイルの記述

すべてのモジュールとチャネルの記述が完了したら，最後にシステムファイルを記述します．サンプル回路のシステムファイルの記述を**リスト7**に示します．システムファイルの記述形式を以下に示します．

```
#include "systemc.h"
//ヘッダファイルのインクルード
int sc_main(int argc, char *argv[])
{
  //チャネルの定義
  //モジュールのインスタンス化
  //ポート接続
  //シミュレーションの記述
  return 0;
}
```

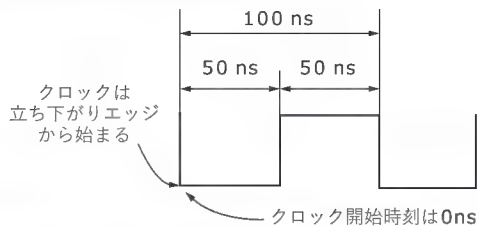
システムファイルでは，SystemCのライブラリ，システム内の全モジュールとチャネルのヘッダファイルをインクルードする必要があります．そして，システム全体のメイン関数であるsc_main()を記述します．sc_main()の内部では，記述したモジュールのインスタンス化，チャネルの定義などによってシステム全体の構成を記述します．なお，最後には必ずreturn 0;を記述します．

▶チャネルの定義

sc_main()内では，まず，各モジュールを接続するためのチャネルを定義します．チャネルの定義は次のようになります．

チャネル名<データ型> チャネルインスタンス名;

今回の例では，サンプル回路の入出力用のチャネルには，

〔図11〕生成されるクロック波形

立ち下がりエッジから始まる周期100nsのクロック波形が生成される．

SystemCの基本チャネルであるsc_signalを用いました．モジュール間の接続のためのsample_channelもここで定義します．

クロックについては，クロック周期などを細かく定義する必要があります．クロックの定義にはいくつかの方法がありますが，ここではサンプル回路での記述例について解説します．**リスト7**の中では，クロックは次のように定義されています．

```
sc_clock clk("clk",100,SC_NS,0.5,0,SC_NS,
                                false);
```

上の記述の引き数は，左側から，名前，クロック周期，クロック周期の時間単位，クロックの比率，スタート時間，スタート時間の単位，最初のクロックエッジの方向，となっています．この例でのクロック波形は**図11**のようになります．ここで，最後の引き数をtrueにすることによって，立ち上がりエッジからクロックを始めることができます．またクロックの比率は，クロック周期に対してクロック信号の値が1となる時間の比率を表しています．クロックの比率を変更することによって，**図12**に示すように非対称なクロック波形も生成することができます．

▶モジュールのインスタンスとポート接続

次に，記述したモジュールをインスタンス化します．モジュールのインスタンス化は次のように記述します．

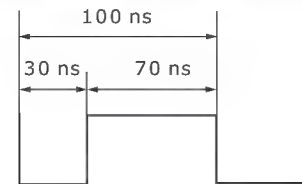
```
モジュール名 インスタンス名("名前");
```

この“ ”で囲まれた名前に関しては，どんな名前でもよいのですが，わかりやすくするためにインスタンス名と一緒にすることをお薦めします．

次に，インスタンス化したモジュールのポートに，先に定義したチャネルを接続します．ポートの接続には二つの方法があります．一つはポート名による接続，もう一つが記述の順番による接続です．以下に，ポート接続の記述を示します．

```
//ポート名による接続
インスタンス名.ポート名(チャネル名) ;
//記述の順番による接続
インスタンス名(チャネル名1,チャネル名2,...)
```

ポート名による接続では，モジュール内に定義したポートの一つ一つに対して，システムファイル内で定義したチャネルインスタンス名を割り当てていきます．一方，記述の順番による

〔図12〕クロックの比率を0.7にした場合に生成されるクロック波形

接続では，モジュール内に記述した順番に，ポートに接続するチャネルインスタンス名を記述していきます．このとき，モジュールのポート名を記述する必要はありません．

どちらの接続方法でも意味は同じになります．記述の順番による接続のほうが少ない記述量ですむため，全体をコンパクトに記述することができます．しかし，接続の記述を誤る可能性も高いので，ポート名による接続で確実に記述したほうが良いでしょう．記述例では，ポート名による接続を行っています．

▶シミュレーションの記述

システムの構成を記述したら，最後にシミュレーションの手順を記述します．シミュレーションの実行は次のように記述します．

```
sc_start(実行時間,単位) ;
```

sc_start()関数は，引き数で指定した時間だけクロックを生成します．なお，sc_start()は，sc_main()の中でしか記述することができません．

サンプル回路の記述例では，switch文で回路に入力する値を選択して，100nsの間シミュレーションを行います．そして，これをfor文で指定回数繰り返すことによって，サンプル回路に連続的に整数データを入力しています．結果として，3000ns間のシミュレーションを行うことになります．

シミュレーション結果は，出力チャネルの値をC++言語と同様に，出力ストリームに出力させることによって，観測することができます．このほか，シミュレーション中の信号(チャネル)の変化を波形トレースとして出力させることもできます．波形トレースを出力させる記述は，以下のようになります．

```
//トレースファイルのファイルポインタの定義
sc_trace_file *ファイルポインタ名;
//トレースファイルの生成
ファイルポインタ名 = sc_create_XXX_trace_file(
                              "トレースファイル名");
//トレース時間単位の設定
((XXX_trace_file *)ファイルポインタ名)->
                  sc_set_XXX_time_unit(-9);
//トレースする信号の登録
sc_trace(ファイルポインタ名,信号名,"信号名");
```

記述中のXXXの部分は，トレースファイルのフォーマットに応じて変更します．サポートしているフォーマットは，VCDフォーマット(sc_vcd_trace)，ISDBフォーマット

〔図13〕記述例の実行結果

〔図14〕トレースファイルの波形

トレースファイルを波形ビューワを用いて観測した結果，サンプル回路が正しく動作していることが確認できる．なお，VCDフォーマットの波形ビューワにはGTKWaveを使用した．GTKWaveは無償で配布されており，以下のURLからダウンロードできる．

(sc_isdb_trace)，WIFフォーマット(sc_wif_trace)の3種類です．各自の開発環境にあったフォーマットを選択してください．サンプル回路の記述例では，VCDフォーマットのトレースファイルを出力します．

以上で，システムファイルの記述が完了します．

● サンプル回路の動作記述

最後に，サンプル回路の動作記述について簡単に説明します．モジュールcheckerでは，クロックごとに入力されるデータを通信関数s_write()を使って，次々にsample_channelチャネルに書き込んでいきます．4クロック目の入力データの値(チェックサム)と計算したチェックサムの値が異なっているときは，通信関数s_reset()を使用してバッファ内のデータを破棄します．したがって，入力されたチェックサムが正しい値とならない限り，sample_channelに書き込まれるデータ数が四つになることはありません．

モジュールaluは，プロセスfetch_decode()がクロックごとにチャネルに格納されているデータの数をチェックします．データの数が四つになったときは，入力されたデータに誤りが含まれていないことになります．このとき，プロセスfetch_decode()は通信関数s_read()を使用して，チャネル内の四つのデータを読み込みます．この時点ではチェックサムのデータは不要ですが，バッファを空にするためにチェックサムも読み込むようにしています．読み込んだデータのうち，一つ目(1クロック目)のデータから実行する演算命令を決定し，演算を実行するプロセスをイベントを使って起動しています．演算終了後は，演算を実行したプロセスが演算結果を出力します．

● SystemCのコンパイル

すべてのファイルを記述したら，最後にコンパイルします．コンパイルはmakeコマンドを使って行います．Makefileの記述に関しては，SystemCをインストールしたディレクトリ(systemc-2.0.1/)内のexample/というディレクトリにSystemCのサンプル記述があるので，そこのMakefileを参考にしてください．

makeコマンドを実行すると，実行ファイルが生成されます．その実行ファイルを実行すると回路のシミュレーション結果が表示されます．また，トレースファイルを出力させていた場合は，実行後にトレースファイルが生成されています．

サンプル回路の記述例を実行した結果とトレースファイルの波形を，それぞれ**図13**と**図14**に示します．**図14**の結果を見ると，正しくサンプル回路が動作していることがわかります．

おわりに

ここまで，簡単ではありますがSystemCの基礎について説明してきました．また，C言語の知識をもっている人なら，SystemCによるシステムの設計やその記述方法についてもある程度理解できたことでしょう．

複雑なシステムを効率良く設計するためには，SystemCのようなシステムレベル設計言語が必要不可欠です．今後，SystemCは多くの開発現場で採用されることになると思います．SystemCによるシステム設計では，ソフトウェアだけでなく，ハードウェア(ディジタル回路)の設計知識も合わせもつことが，より品質の高いシステムを設計するうえで重要になってきます．

以降の章では，SystemCによるさまざまなハードウェア，とくにディジタル回路の記述例を豊富に紹介していきます．そして，SystemCを用いたより実践的なディジタル回路の設計手法について解説していきます．

しまじり・ひろゆき　琉球大学大学院 理工学研究科 総合知能工学専攻
よしだ・たけお　琉球大学 工学部 情報工学

Chapter 2

ディジタル回路設計の基本である
組み合わせ回路を記述する

組み合わせ回路とSystemC記述

吉田たけお

本章からは実際にSystemCでディジタル回路設計を行う．ディジタル回路の基本は，組み合わせ回路と順序回路である．これらは小規模な回路はもちろん，大規模回路を設計するうえでも必要になる基礎概念である．

そこで本章では，まず組み合わせ回路の基礎を学び，それらをSystemCへと置き換えていくことにより，SystemCへの理解を深めていく．

（編集部）

はじめに

本特集では，C言語に関する知識はあるけれど，ハードウェアについてはあまり詳しくない，という方を対象に，SystemCの基礎について解説します．第1章では，SystemCの特徴や記述方法などの概要を説明したので，第2章から第4章では，実際のハードウェア（ディジタル回路）をSystemCで記述する方法について，具体例を示しながら見ていきます．

あとで詳しく解説しますが，ディジタル回路は，組み合わせ回路（combinational circuit）と順序回路（sequential circuit）に大別されます．第2章では前者の組み合わせ回路について，第3章と第4章では後者の順序回路について，それぞれの回路を，SystemCで記述する方法や実際の記述例を示していきます．そこで本章では，まず，ディジタル回路の基礎および組み合わせ回路について簡単に説明します．

1　ディジタル回路とは？

● アナログ回路とディジタル回路

ディジタル回路（digital circuit）とは，一言でいうと，ディジタル信号を扱う電子回路です．

電子回路が扱う電気信号には，**図1**に示すようなアナログ信号とディジタル信号があります．アナログ信号とは，**図1**（**a**）に示すように，時間に対して連続的に電圧値が変化するような電気信号のことをいいます．これに対してディジタル信号とは，**図1**（**b**）に示すように，時間に対して，電圧値が0，1のように離散的に変化するような電気信号のことをいいます．一般に，アナログ信号を扱う電子回路をアナログ回路（analog circuit），ディジタル信号を扱う電子回路をディジタル回路と呼びます．

しかし，アナログ回路もディジタル回路も，それらを構成している部品は同じであり，どちらもトランジスタ，抵抗，コンデンサなどの電子部品を用いて構成されています．すなわち，アナログ回路とディジタル回路は，回路の構造ではなく，回路の機能によって区別されていることになります．そこで，ディジタル信号を「扱う」電子回路について，以下で，もう少し詳しく説明します．

● ディジタル回路の外観

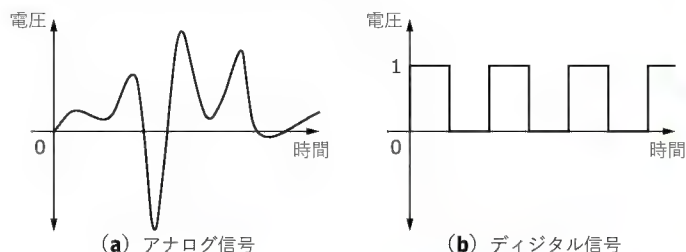
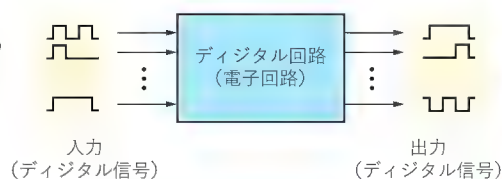
ディジタル回路は，**図2**に示すように，電気信号を入出力するための入力線および出力線をもち，これらの電気信号がディジタル信号になっているような電子回路です．ディジタル回路に入力されたディジタル信号は，何らかの処理が施されます．そして，この処理の結果として得られる新たなディジタル信号が，そのディジタル回路から出力されます．このような意味で，ディジタル回路を，ディジタル信号を「扱う」電子回路と説明しました．

ここでは，ディジタル回路の外観を眺めてみたので，以下では，ディジタル回路の内部を覗いてみましょう．

● ディジタル回路を構成する部品

以上で見たようなディジタル回路は，ゲート（gate）回路と呼

〔図1〕アナログ信号とディジタル信号

（a）アナログ信号　（b）ディジタル信号

〔図2〕ディジタル回路の入力と出力

ばれる単純な機能をもった基本回路を多数組み合わせて構成されます[注1]．ゲート回路には，いくつかの種類があり，**図3**に示すような記号を用いて表されます．**図3**に示した各ゲート回路は，その左側が入力線，右側が出力線になっています．

1）NOTゲート〔**図3(a)**〕

このゲート回路は，入力線と出力線を1本ずつもち，入力Aの0，1を反転した（入れ換えた）値を出力します．値Aを反転した値は$\overline{A}$と表されます．この表現を使うと，$\overline{0}=1$，$\overline{1}=0$ということがわかります．ゲート回路の出力の値は，入力された値によって一意に決まるので，これを関数の表現を用いて，

$$f(A) = \overline{A} \quad \cdots\cdots(1)$$

と表します．このような関数を論理関数（logical function）と呼びます．式(1)は，NOTゲートが実現している論理関数ということになります．なお，式(1)の論理関数で表される演算を否定（negation）ともいいます．

2）ANDゲート〔**図3(b)**〕

このゲート回路は，2本の入力線と1本の出力線をもち，2本の入力A，Bの値がともに1のときのみ1を出力し，それ以外の場合は0を出力します．ANDゲートが実現している論理関数は，論理積（conjunction）とも呼ばれ，

$$f(A,B) = A \cdot B \quad \cdots\cdots(2)$$

と，記号“・”を用いて表されます．記号“・”は，通常，乗算を表しますが，ディジタル回路を扱う場合は，論理積（AND）のことを意味します．

3）ORゲート〔**図3(c)**〕

このゲート回路は，2本の入力線と1本の出力線をもち，2本の入力A，Bの値がともに0のときのみ0を出力し，それ以外の場合は1を出力します．ORゲートが実現している論理関数は，論理和（disjunction）とも呼ばれ，

$$f(A,B) = A+B \quad \cdots\cdots(3)$$

と，記号“＋”を用いて表されます．記号“＋”は，通常，加算を表しますが，ディジタル回路を扱う場合は，論理和（OR）のことを意味します．

4）NANDゲート〔**図3(d)**〕

このゲート回路は，2本の入力線と1本の出力線をもち，2本の入力A，Bの値がともに1のときのみ0を出力し，それ以外の場合は1を出力します．NANDゲートが実現している論理関数は，論理積否定（negative AND）とも呼ばれ，

$$f(A,B) = \overline{A \cdot B} \quad \cdots\cdots(4)$$

のように，論理積（AND）を否定（NOT）した表現になります．

5）NORゲート〔**図3(e)**〕

このゲート回路は，2本の入力線と1本の出力線をもち，2本の入力A，Bの値がともに0のときのみ1を出力し，それ以外の場合は0を出力します．NORゲートが実現している論理関数は，論理和否定（negative OR）とも呼ばれ，

$$f(A,B) = \overline{A+B} \quad \cdots\cdots(5)$$

のように，論理和（OR）を否定（NOT）した表現になります．

6）XORゲート〔**図3(f)**〕

このゲート回路は，2本の入力線と1本の出力線をもち，2本の入力A，Bの値が等しい場合に0を出力し，異なる場合に1を出力します．XORゲートが実現している論理関数は，排他的論理和（exclusive OR）とも呼ばれ，

$$f(A,B) = A \oplus B \quad \cdots\cdots(6)$$

のように，記号“⊕”を用いて表されます．

● ゲート回路の機能の真理値表による表現

以上では，各ゲート回路を回路記号や論理関数を用いて表しましたが，そのゲート回路の入力信号と出力信号の対応表によって表すこともできます．このような対応表は，真理値表（truth table）と呼ばれます．**図3**に示した各ゲート回路の真理値表は，**表1**のようになります．

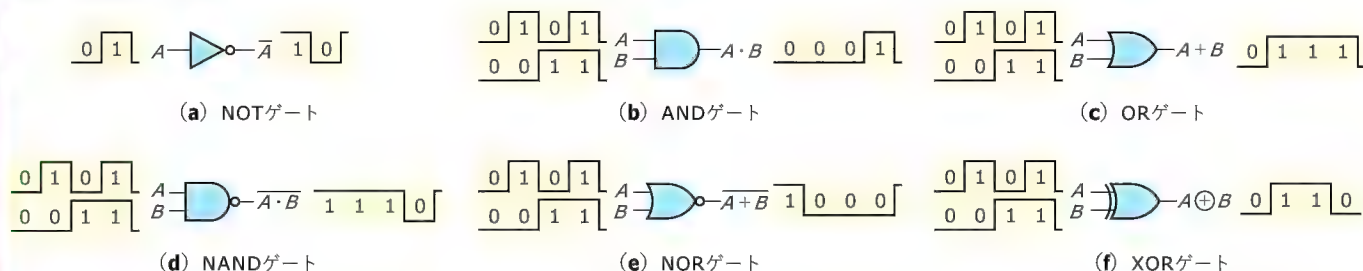
● ゲート回路の内部構造

ここまでは，各ゲート回路を，あたかも実在する一つの電子部品のように，それぞれを一つの記号で表しましたが，各ゲート回路の内部は，トランジスタ，抵抗，コンデンサなどの電子部品で構成さています．すなわち冒頭で述べたように，ディジタル回路は，トランジスタ，抵抗，コンデンサなどの電子部品で構成されていることになります．

以上で説明した各ゲート回路は，それぞれが一つのディジタル回路であると同時に，より大きなディジタル回路を構成するための部品でもあります．ゲート回路は，ディジタル回路の部

〔図3〕各種ゲート回路

(a) NOTゲート　(b) ANDゲート　(c) ORゲート

(d) NANDゲート　(e) NORゲート　(f) XORゲート

注1：ゲート回路を組み合わせて構成される回路は，厳密には，論理回路（logical circuit）と呼ばれている．論理回路はディジタル回路だが，両者は異なる．しかし，ディジタル回路の大部分は，この論理回路で構成されているため，本特集では，これらを区別せずに，単にディジタル回路と呼ぶ．

〔表1〕各種ゲート回路の真理値表

A	$f(A)$
0	1
1	0

(a) NOTゲートの真理値表

A	B	$f(A, B)$
0	0	0
0	1	0
1	0	0
1	1	1

(b) ANDゲートの真理値表

A	B	$f(A, B)$
0	0	0
0	1	1
1	0	1
1	1	1

(c) ORゲートの真理値表

A	B	$f(A, B)$
0	0	1
0	1	1
1	0	1
1	1	0

(d) NANDゲートの真理値表

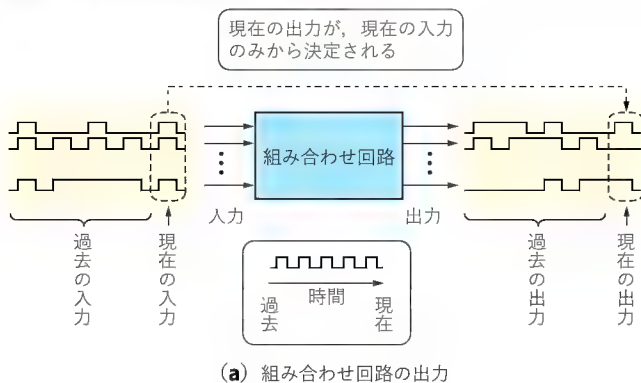
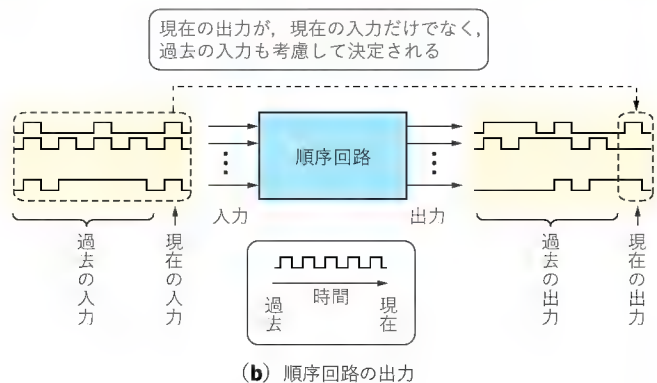
A	B	$f(A, B)$
0	0	1
0	1	0
1	0	0
1	1	0

(e) NORゲートの真理値表

A	B	$f(A, B)$
0	0	0
0	1	1
1	0	1
1	1	0

(f) XORゲートの真理値表

表(a)のNOTゲートの真理値表は，NOTゲートが表している論理関数$f(A) = \overline{A}$を入出力の対応表として表したものである．**表(b)**のANDゲートの真理値表は，ANDゲートが表している論理関数$f(A, B) = A \cdot B$を入出力の対応表として表したものである．**表(c)**のORゲートの真理値表は，ORゲートが表している論理関数$f(A, B) = A+B$を入出力の対応表として表したものである．**表(d)**のNANDゲートの真理値表は，NANDゲートが表している論理関数$f(A, B) = \overline{A \cdot B}$を入出力の対応表として表したものである．**表(e)**のNORゲートの真理値表は，NORゲートが表している論理関数$f(A, B) = \overline{A+B}$を入出力の対応表として表したものである．**表(f)**のXORゲートの真理値表は，XORゲートが表している論理関数$f(A, B) = A \oplus B$を入出力の対応表として表したものである．

〔図4〕組み合わせ回路と順序回路

(a) 組み合わせ回路の出力

(b) 順序回路の出力

品としては，最小単位の部品になります．実際には，ディジタル回路は，いくつかのゲート回路を用いて構成される簡単な機能をもった回路，たとえば，加算器やマルチプレクサなど，を部品として構成されます．加算器やマルチプレクサについては，あとで説明します．

2 組み合わせ回路とは？

● ディジタル回路の分類

先にも述べましたが，ディジタル回路は，組み合わせ回路と順序回路に大別されます．ここで，両者の違いについて説明します．

組み合わせ回路とは，**図4(a)**に示すように，現在の出力が，現在の入力だけから決まるようなディジタル回路です．これに対して順序回路とは，**図4(b)**に示すように，現在の出力が，現在の入力だけでなく，過去の入力も考慮して決定されるようなディジタル回路です．このように，順序回路の出力が過去の入力にも影響を受けるということは，その順序回路内に過去の入力が記憶されている必要があります．この「記憶」という言葉を用いると，組み合わせ回路は記憶機能をもたないディジタル回路，順序回路は記憶機能をもつディジタル回路，と言い換えることもできます．

● 組み合わせ回路の構成

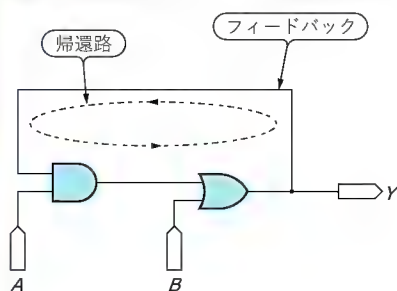
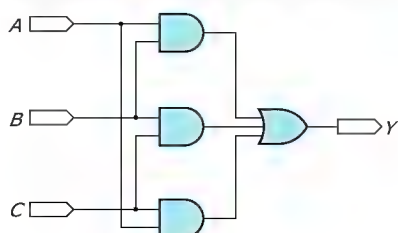
図3に示した各ゲート回路は，記憶機能をもちません．すなわち，各ゲート回路はもっとも簡単な組み合わせ回路ということができます．

ディジタル回路は，このようなゲート回路をいくつか用いて，あるゲート回路の出力線を他のゲート回路の入力線に接続する，という手順で構成されます．1本の出力線を途中で枝分かれさせて，複数の入力線に接続してもかまいません．このとき，上記の手順で得られたディジタル回路に帰還路(feedback loop)がなければ，そのディジタル回路は組み合わせ回路になります．ここで帰還路とは，**図5**に示す回路のように，回路のある地点から出発して，信号線とゲート回路を交互にたどっていったとき，元の地点に戻ってくるような経路のことをいいます．帰還路は，**図5**に示すような，回路の出力側から入力側へ戻るような信号線がある場合に形成されます．このような信号線は，フィードバック(feedback)と呼ばれます．すなわち組み合わせ回路は，フィードバックのないディジタル回路ということもできます．

なお第3章で詳しく説明しますが，フィードバックのある回路は，記憶機能をもってしまう場合があります．このフィードバックを利用したもっとも単純な記憶回路にフリップフロップ(flip-flop：FF)がありますが，これは順序回路を構成する際に用いる回路なので，次章で説明します．

● 組み合わせ回路の表現

ここでは，組み合わせ回路の表現方法について見ていきます．

〔図5〕帰還路をもつディジタル回路の例

〔図6〕組み合わせ回路の例(多数決回路)

〔表2〕多数決回路の真理値表

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

入力 A, B, C のうち,1となる信号線の数が二つ以上の行で出力 Y が1,一つ以下の行で出力 Y が0になっているので,多数決回路の真理値表であることが確認できる.

まず,**図6**の回路について考えてみましょう.この回路は,フィードバックがないので,組み合わせ回路になっています.**図6**では,3入力のORゲートが使われていますが,これは,三つの入力すべてが0の場合に0を出力し,それ以外の場合に1を出力する回路です.3入力のORゲートは,2入力のORゲートを2個接続することで作ることができます.

この回路がどのような機能をもっているか考えてみましょう.この回路は,3個のANDゲートと1個のORゲートで構成されています.3個のANDゲートは,A, B がともに1,B, C がともに1,C, A がともに1,の場合にそれぞれ1になり,それ以外の場合は0になります.ORゲートは,これらのANDゲートの出力がすべて0になる場合のみ0を出力し,それ以外の場合に1を出力します.すなわち,**図6**の回路は,A, B, C の三つのうち,1となる信号線の数が二つ以上の場合に1を出力し,一つ以下の場合に0を出力する回路になっています.このような回路は,多数決回路(voter)と呼ばれます.

組み合わせ回路は,すべての入力の組み合わせに対して,そのときの出力を求めることによって,回路全体の真理値表を作ることができます.**図6**に示した多数決回路の真理値表は,**表2**のようになります.この真理値表からも,出力の値が,入力の値の多数決結果になっていることが確認できます.

さらに,多数決回路が実現している機能を論理関数として表すこともできます.具体的には,多数決回路は,以下に示す論理関数を実現しています.

$$Y = f(A, B, C) = (A \cdot B) + (B \cdot C) + (C \cdot A) \quad \cdots\cdots\cdots\cdots(7)$$

この例で見たように,組み合わせ回路の機能を表現する方法には,ゲート回路の場合と同様に,回路図による表現,真理値表による表現,論理関数による表現があることがわかります.

これらの表現はそれぞれ等価な表現なので,組み合わせ回路を設計する際は,場合に応じて,都合の良い表現を用いることができます.

3 組み合わせ回路の設計例

● シミュレーション環境について

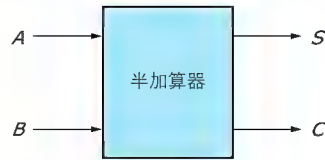
第2章から第4章では,よく用いられるディジタル回路とそのSystemCによる記述を紹介します.そこでまず,以下で用いるシミュレーション環境について,簡単に説明しておきます.

SystemCのコンパイラは,OSCI(Open SystemC Initiative)で配布しているSystemC 2.0.1[注2]を使用しています.また,シミュレーション結果の波形を表示させるために,GTKWave 2.0.0pre1[注3]を使用しています.これらは,いずれも無償で利用可能です.なお,GTKWaveは,VCD(value change dump)と呼ばれる形式のトレースファイルを波形として表示するためのツール(波形ビューア)です.これらの使用法に関しては,付属のマニュアルなどを参照してください.

なお,第2章から第4章で示すSystemCの記述例はすべて,Verilog-HDLやVHDLなどのハードウェア記述言語(Hardware Description Language:HDL)への変換が可能なRTL(Register Transfer Level)モデルの記述になっています.HDLによる回路の記述は,回路図への自動変換が可能です.このHDL記述を回路図へ変換することを論理合成(logic synthesis)または単に合成(synthesis)といい,論理合成を自動的に行うソフトウェアを論理合成ツールといいます.第2章から第4章では,SystemC記述を実際にVerilog-HDLに変換[注4]し,そのVerilog-HDL記述を論理合成ツールを用いて合成した結果も示します.

SystemC記述をVerilog-HDL記述へ変換するために,米国Summit Design社のVisual Elite Ver 3.1.3を使用しました[注5].また,論理合成ツールとして,米国Synopsys社のDesign Compiler Ver.2003.03を使用しました.これらのソフトウェアは無償ではありませんが,Visual Elite Ver 3.1.4の評価版が本

注2:OSCIのWebサイト(http://www.systemc.org/)でユーザー登録をしてから,ダウンロードできる.
注3:http://www.sc.man.ac.uk/apt/tools/gtkwave/index.html からダウンロードできる.
注4:誌面の都合により,変換後のVerilog-HDL記述は割愛した.

〔図7〕半加算器の入力と出力

〔表3〕半加算器の真理値表

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

1桁の2進数A，Bの和Sは，A，Bの一方が1で他方が0の場合に1となり，それ以外の場合に0となる．このため，和Sは，A，Bの排他的論理和(XOR)となる．またこのとき，キャリが生じるのは，A，Bがともに1となる場合のみである．このため，キャリCは，A，Bの論理積(AND)となる．

号付属CD-ROMに収録されています．また，論理合成ツールに関しても，今回使用したツールとは異なりますが，本誌の姉妹誌Design Wave Magazine 2003年4月号の付属CD-ROMに評価版が収録されています．興味をもたれた方は，ぜひ使用してみてください．

● 半加算器の設計

ディジタル回路の部品としてよく用いられる組み合わせ回路の一つに，加算や乗算を行う演算器が挙げられます．ディジタル回路では主として2値データが扱われるので，演算器で演算される値も2進数になります．もっとも基本的な演算器は，1ビット(1桁)の2進数同士の加算を行う演算器です．このような回路を半加算器(half adder)と呼びます．

1ビットの2進数同士の加算結果は，桁上げ(キャリ：carry)が生じて2ビットになる可能性があります．そのため半加算器は，**図7**に示すように，入力線と出力線をそれぞれ2本ずつもつ回路になります．また，**表3**に示すように，1ビットの2進数A，Bの加算結果を表にまとめると，半加算器の真理値表になります．なお**表3**において，出力SはA，Bの和(の1桁目)を，出力Cはキャリをそれぞれ表しています．

表3を見るとわかりますが，和Sの真理値表は，XORゲートの真理値表と一致しています．同様に，キャリCの真理値表は，ANDゲートの真理値表と一致しています．すなわち半加算器は，XORゲートとANDゲートを1個ずつ用いて構成できます．

第1章で見たように，SystemCでは，個々のシステムをヘッダファイルとインプリメンテーションファイルで，また，システムの最上位階層をシステムファイルで，それぞれ記述します．まず，半加算器のヘッダファイルとインプリメンテーション

〔リスト1〕半加算器のヘッダファイルの記述例(half_adder.h)

```
#ifndef __HALF_ADDER__
#define __HALF_ADDER__

#include "systemc.h"

SC_MODULE(half_adder){
  sc_in<bool>  A;
  sc_in<bool>  B;
  sc_out<bool> S;
  sc_out<bool> C;

  void half_adder_rtl(void);

  SC_CTOR(half_adder){
   SC_METHOD(half_adder_rtl);
    sensitive << A << B;
  }
};

#endif
```

半加算器には，入力ポートA，Bと出力ポートS，Cがあることを宣言し，Methodプロセスは，A，Bの少なくとも一方の値が変化したときに起動することを宣言している．

〔リスト2〕半加算器のインプリメンテーションファイルの記述例1(half_adder.cpp)

```
#include "half_adder.h"

void half_adder::half_adder_rtl(void){
  bool SIG_A;
  bool SIG_B;
  bool SIG_S;
  bool SIG_C;

  SIG_A = A.read();
  SIG_B = B.read();

  SIG_S = SIG_A ^ SIG_B;
  SIG_C = SIG_A & SIG_B;

  S.write(SIG_S);
  C.write(SIG_C);
};
```

信号Sが，信号AとBの排他的論理和(XOR)で，信号Cが，信号AとBの論理積(AND)であることを定義している．

ファイルをそれぞれ**リスト1**，**リスト2**に示します．

▶**ヘッダファイルの記述内容**

まず，ヘッダファイルを上から順に見てみましょう．ファイルの最初と最後の#ifndef～#endifは，ヘッダファイルの2度読みによるエラーを防止するための記述です．

次の，#include "systemc.h"は，インプリメンテーションファイルに記述しておいてもかまいませんが，SystemCを使用するためのライブラリなので，必ず記述してください．

この次からが，ヘッダファイルの内容になります．ヘッダファイルでは，モジュール(設計する回路)およびそのモジュールがもつポート(入出力線)の情報，すなわち，回路の外観を記述します．これに対し，インプリメンテーションファイルでは，回路の動作をプロセスとして記述します．このプロセスの起動条件もヘッダファイルに記述します．この起動条件は，そのプロセスに対する入力信号のうち，どの信号が変化したらプロセスを起動するのかを，“センシティビティリスト”というもので指定します．第3章で詳しく説明しますが，組み合わせ回路の場合，プロセスに対する入力信号はすべてセンシティビティリストに記述します．

注5：Visual Eliteは，システムレベルの設計および検証作業を支援する統合設計環境であり，SystemC記述をHDL記述へ変換する機能は，Visual Eliteがもつ機能のほんの一部である．なおVisual Eliteは，Summit Design Japan社のご厚意により，使用の許可をいただいた．

このヘッダファイルでは，half_adderという名前(回路名)のモジュールを定義しています．また，半加算器の外観は，**図7**で見たとおりで，その入出力線が，そのままポートとして定義されています．なお，SystemCでは，1ビットのデータ型として，bool型が使用できます．さらに，このhalf_adderで実行するプロセスをhalf_adder_rtlという名前で定義し，このプロセスが，信号*A*，*B*の少なくとも一方が変化した場合に，起動されることをコンストラクタ内のセンシティビティリストで指定しています．以上が，半加算器のヘッダファイルの内容になります．

〔リスト3〕半加算器のインプリメンテーションファイルの記述例2 (half_adder.cpp)

```
#include "half_adder.h"

void half_adder::half_adder_rtl(void){
  S = A ^ B;
  C = A & B;
};
```

信号*S*が，信号*A*と*B*の排他的論理和(XOR)で，信号*C*が，信号*A*と*B*の論理積(AND)であることを定義している．

〔リスト4〕半加算器のシステムファイルの記述例 (main_half_adder.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "half_adder.h"

int sc_main(int argc, char *argv[]){
  int i;
  sc_signal<bool> A;
  sc_signal<bool> B;
  sc_signal<bool> S;
  sc_signal<bool> C;

  half_adder HA("half_adder");

  HA.A(A);
  HA.B(B);
  HA.S(S);
  HA.C(C);

  sc_trace_file *trace_f;
  trace_f = sc_create_vcd_trace_file("half_adder_trace");

  ((vcd_trace_file *)trace_f)->sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,A,"A");
  sc_trace(trace_f,B,"B");
  sc_trace(trace_f,S,"S");
  sc_trace(trace_f,C,"C");

  cout << "A B | S C" << endl;
  cout << "---------" << endl;
  for(i=0;i<20;i++){
    if(i%2)     A = true; else A = false;
    if((i/2)%2) B = true; else B = false;
    sc_start(50,SC_NS);
    cout << A <<" " << B <<" | "<< S <<" " << C << endl;
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

信号*A*には，50[ns]ごとに0(false)，1(true)が交互に入力され，信号*B*には，100[ns]ごとに0，1が交互に入力される．

▶インプリメンテーションファイルの記述内容

次に，インプリメンテーションファイルについて見てみましょう．インプリメンテーションファイルでは，先に触れたように，回路の動作をプロセスとして記述します．半加算器は，XORゲートとANDゲートで実現できるので，**リスト2**のようになります．この記述例は，非常に簡単ですが，より複雑な動作を記述する場合には，if文，switch文，for文などの構文を使用することも可能です．

なお，**リスト2**では，入力信号を.read()で呼び出し，プロセス内で定義した変数に格納しています．また，.write()を使って，変数の値を出力信号に渡しています．第1章で説明したように，**リスト2**の記述は，**リスト3**に示すような簡単な記述にすることもできますが，以降では，.read()と.write()を用いて記述します．

▶システムファイルの記述内容

以上で回路の記述は完成しましたが，その動作を確認するためには，システムファイルが必要になります．半加算器のシステムファイルの記述例を**リスト4**に示します．システムファイルの記述方法は，第1章で説明したとおりなので，そちらを参照してください．なお，**リスト1**，**リスト2**，**リスト4**をコンパイルして得られる実行ファイルを実行すると，half_adder_trace.vcdという名前のファイルが作成されます．このファイルは，トレースファイルと呼ばれ，シミュレーション結果の情報が記録されています．このトレースファイルをGTKWaveを使って波形表示すると，**図8**のようになります．この波形から，**リスト1**，**リスト2**の記述が半加算器になっていることが確認できます．

なお，**リスト1**，**リスト2**の記述をVisual EliteでVerilog-HDLに変換し，さらに，その記述をDesign Compilerで論理合成した結果を，**図9**に示します．**図9**では，最初に確認したとおり，半加算器が，XORゲートとANDゲートから構成されることがわかります．

● 全加算器の設計

半加算器は，二つの1ビット2進数の和を求める回路だったので，今度は，二つの*n*ビット2進数の和を求める回路について考えてみましょう．2進数の加算も10進数の加算と同じルールで計算できます．すなわち，最下位桁から順に，下位桁からのキャリ(桁上げ)を考慮しながら計算していけば，*n*ビットの2進数同士の和を求めることができます．

最下位桁では，それより下位の桁がないので，二つの1ビット2進数の和を求めるだけになります．一方，最下位でない桁では，二つの1ビット2進数のほかに，下位桁からのキャリも考慮する必要があります．すなわち，最下位でない桁では，三つの1ビット2進数の和を求める必要があります．このような，三つの1ビット2進数の和を求める組み合わせ回路を全加算器(full adder)と呼びます．

三つの1ビット2進数の和の最大値は，11(10進数で3)とな

〔図 8〕半加算器のシミュレーション波形

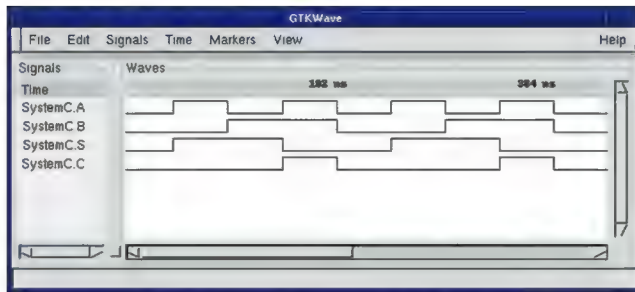



表 3 の真理値表と同じ機能を実現していることが確認できる.

るため，2 ビットで表現できます．すなわち，全加算器は，3 本の入力と 2 本の出力をもった，**図 10** に示すような外観になります．また，全加算器の真理値表は，**表 4** のようになります．なお，**図 10** および**表 4** において，出力 SUM は加算結果の 1 桁目を，出力 COUT は加算結果の 2 桁目を，それぞれ表しています．

ところで余談になりますが，三つの 1 ビット 2 進数の加算においてキャリが生じるのは，三つのうち二つ以上が 1 になる場合です．すなわち，キャリ COUT を求める回路は，先に紹介した多数決回路と同じ機能をもった回路になります．事実，**表 4** の出力 COUT の真理値表は，**表 2** に示した多数決回路の真理値表と一致しています．ディジタル回路では，この例のように，同じ機能をもった回路でも用途によって呼び方が異なる場合があります．すでに設計済みの回路を再利用 (reuse) できるといった場合がよくあるので，設計中の回路をよく見直してみましょう．

さて，本題に戻って，全加算器の設計を行いましょう．ここで全加算器の機能を見直すと，じつは全加算器は，**図 11** に示すように，二つの半加算器と一つの OR ゲートを用いて構成できることがわかります．半加算器は，すでに設計済みなので，今回は，**図 11** のような構成の全加算器を設計してみましょう．このように，設計済みの回路を再利用して，新たな大きな回路を設計することを階層設計 (hierarchical design) といいます．

まず，全加算器のヘッダファイル，インプリメンテーションファイルを，それぞれ**リスト 5**，**リスト 6** に示します．**図 11** の点線で示したように，SystemC では，設計済みのモジュールの呼び出し (インスタンス化) は，ヘッダファイルのコンストラクタ内で行い，残りの部分をプロセスとして記述します．

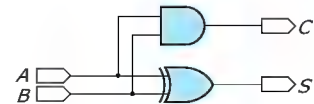
モジュールを呼び出すためには，まず，ヘッダファイル内で，モジュールへのポインタを登録します．

```
half_adder *COMP1;
half_adder *COMP2;
```

このとき，モジュール間の接続に使用する内部信号線も sc_signal として定義しておきます．

```
sc_signal<bool> C1_S, C1_C, C2_C;
```

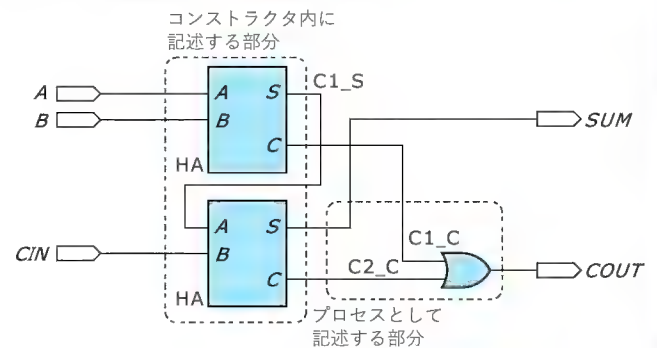
ここで，内部信号線とは，モジュール (回路) の内部でのみ使用される信号線のことをいいます．**図 11** では，C1_S，C1_C，

〔図 9〕
半加算器の合成結果

〔図 10〕
全加算器の入力と出力

〔表 4〕全加算器の真理値表

A	*B*	*CIN*	*SUM*	*COUT*
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

三つの 1 ビット 2 進数の加算表になっており，出力 *SUM* は加算結果の 1 桁目を，出力 *COUT* は加算結果の 2 桁目を，それぞれ表している．

〔図 11〕半加算器による全加算器の構成

C2_C の三つの信号線が入力でも出力でもなく，内部でのみ用いられる信号線なので，内部信号線となります．

そして，たとえば COMP1 の半加算器を呼び出すには，コンストラクタ内で，

```
COMP1 = new half_adder("COMP1");
COMP1->A(A);
COMP1->B(B);
COMP1->S(C1_S);
COMP1->C(C1_C);
```

のように記述します．まず 1 行目で，new 演算子を用いて COMP1 の半加算器をインスタンス化しています．次に 2～5 行目で，COMP1 の半加算器の入出力線に信号線を接続しています．これは，「COMP1 の入力ポート *A* に，全加算器の入力ポート *A*」を，「COMP1 の入力ポート *B* に，全加算器の入力ポート

B」を，「COMP1の出力ポート*S*に，全加算器の内部信号線C1_S」を，「COMP1の出力ポート*C*に，全加算器の内部信号線C1_C」を，それぞれ接続することを表しています．COMP2についても同様に記述します．

二つの半加算器を呼び出したら，残りはORゲートのみとなるので，このORゲートをプロセスとして記述します．このプロセス（ORゲート）に対する入力は，**図11**に示したようにC1_C，C2_Cの二つなので，これらの信号線をセンシティビティリストに指定します．

```
SC_METHOD(carry_rtl);
sensitive << C1_C << C2_C;
```

以上がヘッダファイルの内容になります．なお，**リスト6**に示したインプリメンテーションファイルでは，ORゲートのみを記述しています．

全加算器のシステムファイルの記述例を**リスト7**に，シミュレーション結果を**図12**に示します．**図12**から，全加算器として動作していることが確認できます．また，**リスト5**，**リスト6**をVerilog-HDL記述に変換し，それを論理合成した結果を**図13**に示します．

なお，ここで設計した全加算器をn−1個と，先に設計した半加算器を1個用いることにより，nビットの2進数同士の加算を行う回路を簡単に設計できます．たとえば，4ビットの加算器は**図14**のようになります．この加算器は，筆算による加算の手順をそのまま回路として表した構成になっています．

ディジタル回路では，加算器以外にもさまざまな演算器が用いられますが，以下では，演算器以外でよく用いられる回路を見てみましょう．

● マルチプレクサの設計

複数のデータ入力線から制御信号によって一つの入力線を選

〔リスト5〕全加算器のヘッダファイルの記述例（full_adder.h）

```
#include "systemc.h"
#include "half_adder.h"

SC_MODULE(full_adder){
  sc_in<bool>  A;
  sc_in<bool>  B;
  sc_in<bool>  CIN;
  sc_out<bool> SUM;
  sc_out<bool> COUT;

  sc_signal<bool> C1_S, C1_C, C2_C;

  half_adder *COMP1;
  half_adder *COMP2;

  void carry_rtl(void);

  SC_CTOR(full_adder){
    COMP1 = new half_adder("COMP1");
    COMP1->A(A);
    COMP1->B(B);
    COMP1->S(C1_S);
    COMP1->C(C1_C);

    COMP2 = new half_adder("COMP2");
    COMP2->A(C1_S);
    COMP2->B(CIN);
    COMP2->S(SUM);
    COMP2->C(C2_C);

    SC_METHOD(carry_rtl);
    sensitive << C1_C << C2_C;
  }
};
```

〔リスト6〕全加算器のインプリメンテーションファイルの記述例（full_adder.cpp）

```
#include "full_adder.h"

void full_adder::carry_rtl(void){
  bool SIG_COUT;

  SIG_COUT = C1_C.read() | C2_C.read();

  COUT.write(SIG_COUT);
};
```

〔リスト7〕全加算器のシステムファイルの記述例（main_full_adder.cpp）

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "full_adder.h"

int sc_main(int argc, char *argv[]){
  int i;
  sc_signal<bool> A;
  sc_signal<bool> B;
  sc_signal<bool> CIN;
  sc_signal<bool> SUM;
  sc_signal<bool> COUT;

  full_adder FA("full_adder");

  FA.A(A);
  FA.B(B);
  FA.CIN(CIN);
  FA.SUM(SUM);
  FA.COUT(COUT);

  sc_trace_file *trace_f;
  trace_f = sc_create_vcd_trace_file("full_adder_trace");
  ((vcd_trace_file *)trace_f)->sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,A,"A");
  sc_trace(trace_f,B,"B");
  sc_trace(trace_f,CIN,"CIN");
  sc_trace(trace_f,SUM,"SUM");
  sc_trace(trace_f,COUT,"COUT");

  cout << "A  B CIN | SUM COUT" << endl;
  cout << "-------------------" << endl;
  for(i=0;i<50;i++){
    if(i%2)      A   = true; else  A   = false;
    if((i/2)%2)  B   = true; else  B   = false;
    if((i/4)%2)  CIN = true; else CIN = false;
    sc_start(50,SC_NS);
    cout << A <<"  "<< B <<"  "<< CIN <<
         "  |  "<< SUM <<"   "<< COUT << endl;
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

信号*A*には，50[ns]ごとに0(false)，1(true)が交互に，信号*B*には，100[ns]ごとに0，1が交互に，信号*CIN*には，200[ns]ごとに0，1が交互に，それぞれ入力される．

択し，その信号の値を出力線に伝えるような組み合わせ回路を，マルチプレクサ(multiplexer)と呼びます．マルチプレクサは，データセレクタ(data selector)あるいは単にセレクタ(selector)などとも呼ばれます．

たとえば，4ビットのデータ入力線をもつ4ビットマルチプレクサの外観は，**図15**のようになり，その真理値表は**表5**のようになります．

ここで，**図15**および**表5**に示すように，たとえばn本の信号線をまとめて一つの信号線Aとして表す場合，通常，$A[n-1:0]$と表記します．この表記により，$A[n-1]$，$A[n-2]$，…，$A[0]$の合計n本の1ビットの信号線をまとめて表したことになります．

4ビットマルチプレクサのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト8**，**リスト9**に示します．まず，**リスト8**のヘッダファイルでは，複数の信号線をsc_uint型を用いて定義しています．これは，符号なしの整数型を表します．符号あり整数の場合はsc_int型を用います．**リスト8**では，入力D，Sが，それぞれ4ビット，2ビットの符号なし整数であることを表しています．

一方，**リスト9**のインプリメンテーションファイルでは，if文を用いて，信号線Sの値で場合分けをして，マルチプレクサの出力を決定しています．このように，基本的な構文を用いて回路を記述することも可能です．

マルチプレクサのシステムファイルの記述例を**リスト10**に，シミュレーション結果を**図16**に示します．**図16**から，マルチプレクサとして動作していることが確認できます．また，**リスト8**，**リスト9**をVerilog-HDL記述に変換し，それを論理合成した結果を**図17**(p.69)に示します．

● デマルチプレクサの設計

マルチプレクサとは逆に，制御信号によって，複数の出力線の中から一つの出力線を選択し，データ入力線の値を選択された出力線に伝える組み合わせ回路をデマルチプレクサ(demultiplexer)と呼びます．

たとえば，4ビットの出力線をもつ4ビットデマルチプレクサの外観は，**図18**(p.69)のようになり，また，その真理値表は，**表6**(p.69)のようになります．

4ビットデマルチプレクサのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト11**(p.69)，**リスト12**(p.69)に示します．

リスト12に示したインプリメンテーションファイルでは，マルチプレクサの場合と同様に，if文を用いて，信号線Sの値で場合分けをして，デマルチプレクサの出力を決定しています．

また，デマルチプレクサのシステムファイルの記述例を**リスト13**(p.69)に，シミュレーション結果を**図19**(p.70)に示します．**図19**から，デマルチプレクサとして動作していることが

〔図12〕全加算器のシミュレーション波形

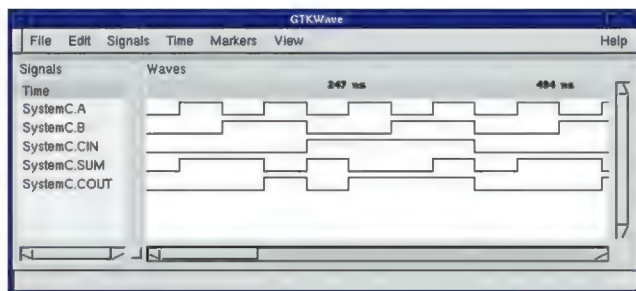
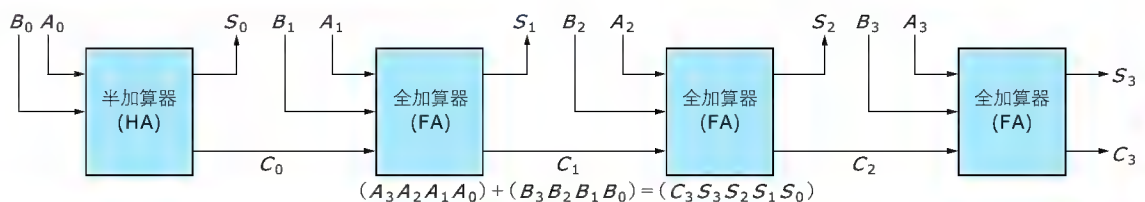
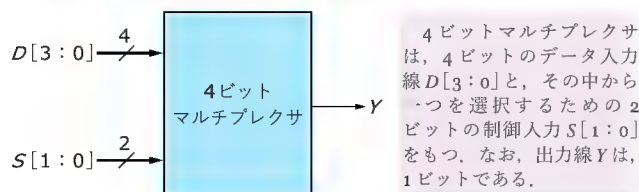
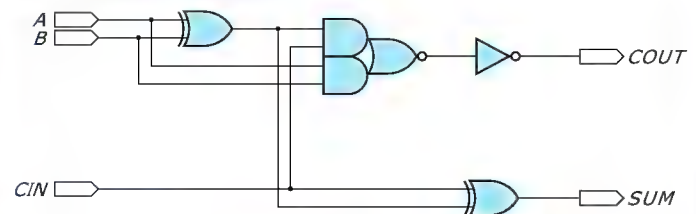



表4の真理値表と同じ機能を実現していることが確認できる．

〔図13〕全加算器の合成結果

〔図14〕4ビット加算器の構成

nビット加算器は，1個の半加算器と$n-1$個の全加算器を用いて，図のように接続することにより実現できる．なお，加算結果は，$n+1$ビットとなる．

〔図15〕4ビットマルチプレクサの入力と出力

4ビットマルチプレクサは，4ビットのデータ入力線$D[3:0]$と，その中から一つを選択するための2ビットの制御入力$S[1:0]$をもつ．なお，出力線Yは，1ビットである．

〔表5〕4ビットマルチプレクサの真理値表

制御入力 S_1	S_0	制御入力Sの10進数の値	出力 Y
0	0	0	D_0
0	1	1	D_1
1	0	2	D_2
1	1	3	D_3

4ビットマルチプレクサでは，制御入力$S[1:0]$の10進数としての値がiであるとき，出力線Yに，$D[i]$の値を出力する．

〔リスト 8〕マルチプレクサのヘッダファイルの記述例（mux.h）

```
#include "systemc.h"

SC_MODULE(mux){
  sc_in<sc_uint<4> > D;
  sc_in<sc_uint<2> > S;
  sc_out<bool>       Y;

  void mux_rtl(void);

  SC_CTOR(mux){
    SC_METHOD(mux_rtl);
    sensitive << D << S;
  }
};
```

n ビットの信号線を定義する場合は，sc_uint<n> を用いる．

〔リスト 9〕マルチプレクサのインプリメンテーションファイルの記述例（mux.cpp）

```
#include "mux.h"

void mux::mux_rtl(void){
  sc_uint<4> TMP_D;
  bool       TMP_Y;

  TMP_D = D.read();

  if ( S.read() == 0x0 ) {
    TMP_Y = TMP_D[0];
  } else if ( S.read() == 0x1 ) {
    TMP_Y = TMP_D[1];
  } else if ( S.read() == 0x2 ) {
    TMP_Y = TMP_D[2];
  } else {
    TMP_Y = TMP_D[3];
  }

  Y.write(TMP_Y);
};
```

if 文を用いて，信号線 *S* の値で場合分けをし，マルチプレクサの機能を記述している．

〔リスト 10〕マルチプレクサのシステムファイルの記述例（main_mux.cpp）

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "mux.h"

int sc_main(int argc, char *argv[]){
  int i;
  sc_signal<sc_uint<4> > D;
  sc_signal<sc_uint<2> > S;
  sc_signal<bool>        Y;

  sc_uint<4> TMP_D;
  sc_uint<2> TMP_S;

  mux MUX("mux");

  MUX.D(D);
  MUX.S(S);
  MUX.Y(Y);

  sc_trace_file *trace_f;
  trace_f = sc_create_vcd_trace_file("mux_trace");

  ((vcd_trace_file *)trace_f)->sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,D,"D");
  sc_trace(trace_f,S,"S");
  sc_trace(trace_f,Y,"Y");

  cout << "D S | Y " << endl;
  cout << "---------" << endl;
  TMP_D = 0x0;
  TMP_S = 0x0;
  for(i=0;i<200;i++){
    if(i>0)     TMP_S++;
    if(!(i%4)) TMP_D++;
    D = TMP_D;
    S = TMP_S;
    sc_start(50,SC_NS);
    cout <<  D <<" " << S <<" | "<< Y << endl;
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

信号 *S* には，50[ns] ごとに 00，01，10，11 が順番に入力され，これが繰り返される．また，信号 *D* には，200[ns] ごとに 0000，0001，…，1111 が順番に入力され，同じく，これが繰り返される．

〔図 16〕マルチプレクサのシミュレーション波形

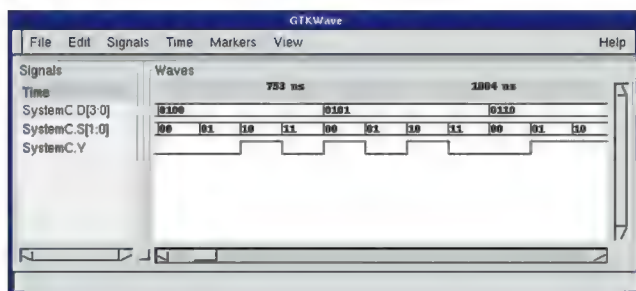



表 5 の真理値表と同じ機能を実現していることが確認できる．

確認できます．さらに，**リスト 11**，**リスト 12** を Verilog-HDL 記述に変換し，それを論理合成した結果を**図 20**（p.70）に示します．

● デコーダの設計

n ビットの信号が入力されると，その *n* ビット 2 進数に対応する番号の出力線のみを 1 にし，残りの出力線を 0 にする組み合わせ回路をデコーダ（decoder）と呼びます．より一般的には，何らかの符号化が施された信号を元に戻す回路を総称してデコーダと呼びます．

たとえば，2 ビットの入力をもつ 2 入力 4 出力デコーダの外観は，**図 21**（p.70）のようになり，また，その真理値表は，**表 7**（p.70）のようになります．

なお，4 ビットデマルチプレクサの入力 *D* を 1 に固定し，制御入力 *S* を 2 入力 4 出力デコーダの入力 *D* とみなすと，その 4 ビットデマルチプレクサは，2 入力 4 出力デコーダの機能を実現します．そのため，デマルチプレクサの SystemC 記述に若干の修正を加えれば，デコーダの記述になります．しかし，他の構文の使用例を見てもらうために，ここでは，case 文を用いてデコーダを記述します．

2 入力 4 出力デコーダのヘッダファイルとインプリメンテー

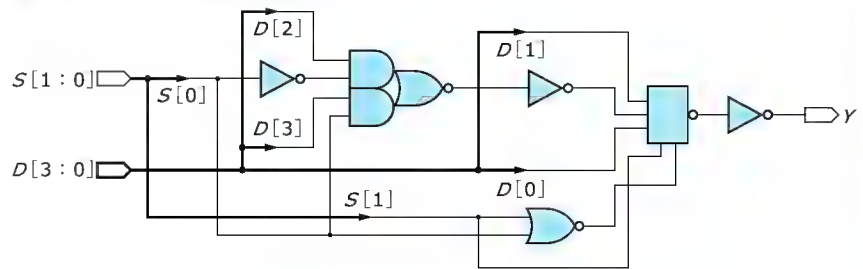
ションファイルを，それぞれ**リスト 14**，**リスト 15** に示します．

リスト 15 に示したインプリメンテーションファイルでは，case 文を用いて，信号線 D の値で場合分けをして，デコーダの出力を決定しています．

また，デコーダのシステムファイルの記述例を**リスト 16** に，シミュレーション結果を**図 22**（p.71）に示します．**図 22** から，デコーダとして動作していることが確認できます．さらに，**リスト 14**，**リスト 15** を Verilog-HDL 記述に変換し，それを論理合成した結果を**図 23**（p.71）に示します．

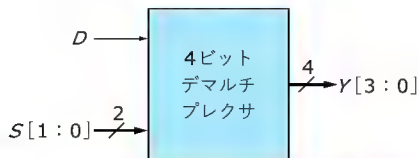
● エンコーダの設計

デコーダとは逆に，何番目の入力線に信号が入ったかを 2 進

〔図 17〕マルチプレクサの合成結果

〔表 6〕4 ビットデマルチプレクサの真理値表

制御入力		制御入力 S の	出力			
S_1	S_0	10 進数の値	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	D
0	1	1	0	0	D	0
1	0	2	0	D	0	0
1	1	3	D	0	0	0

4 ビットデマルチプレクサでは，制御入力 $S[1:0]$ の 10 進数としての値が i であるとき，入力 D の値を出力線 $Y[i]$ から出力する．他の出力線からは，すべて 0 が出力される．

〔図 18〕4 ビットデマルチプレクサの入力と出力

4 ビットデマルチプレクサは，4 ビットの出力線 $Y[3:0]$ と，その中から一つを選択するための 2 ビットの制御入力 $S[1:0]$ をもつ．なお，データ入力線 Y は，1 ビットである．

〔リスト 11〕デマルチプレクサのヘッダファイルの記述例（demux.h）

```
#include "systemc.h"

SC_MODULE(demux) {
  sc_in<bool>          D;
  sc_in<sc_uint<2> >   S;
  sc_out<sc_uint<4> >  Y;

  void demux_rtl(void) ;

  SC_CTOR(demux) {
    SC_METHOD(demux_rtl) ;
    sensitive << D << S;
```

n ビットの信号線を定義する場合は，sc_uint<n> を用いる．

〔リスト 12〕デマルチプレクサのインプリメンテーションファイルの記述例（demux.cpp）

```
#include "demux.h"

void demux::demux_rtl(void) {
  sc_uint<4> TMP_Y;

  TMP_Y = 0x0;

  if ( S.read() == 0x0 ) {
    TMP_Y[0] = D.read() ;
  } else if ( S.read() == 0x1 ) {
    TMP_Y[1] = D.read() ;
  } else if ( S.read() == 0x2 ) {
    TMP_Y[2] = D.read() ;
  } else {
    TMP_Y[3] = D.read() ;
  }

  Y.write(TMP_Y) ;
};
```

〔リスト 13〕デマルチプレクサのシステムファイルの記述例（main_demux.cpp）

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "mux.h"

int sc_main(int argc, char *argv[]) {
  int i;
  sc_signal<sc_uint<4> > D;
  sc_signal<sc_uint<2> > S;
  sc_signal<bool>        Y;

  sc_uint<4> TMP_D;
  sc_uint<2> TMP_S;

  mux MUX("mux") ;

  MUX.D(D) ;
  MUX.S(S) ;
  MUX.Y(Y) ;

  sc_trace_file *trace_f;
  trace_f = sc_create_vcd_trace_file("mux_trace") ;
  ((vcd_trace_file *) trace_f) ->sc_set_vcd_time_unit(-9) ;

  sc_trace(trace_f,D,"D") ;
  sc_trace(trace_f,S,"S") ;
  sc_trace(trace_f,Y,"Y") ;

  cout << "D S | Y " << endl;
  cout << "--------" << endl;
  TMP_D = 0x0;
  TMP_S = 0x0;
  for(i=0;i<200;i++) {
    if(i>0)      TMP_S++;
    if(!(i%4))  TMP_D++;
    D = TMP_D;
    S = TMP_S;
    sc_start(50,SC_NS) ;
    cout <<  D <<" " << S <<" | "<< Y << endl;
  }

  sc_close_vcd_trace_file(trace_f) ;

  return 0;
}
```

信号 S には，50[ns] ごとに 00，01，10，11 が順番に入力され，これが繰り返される．また，信号 D には，100[ns] ごとに 0(false)，1(true) が交互に入力される．

〔図 19〕デマルチプレクサのシミュレーション波形

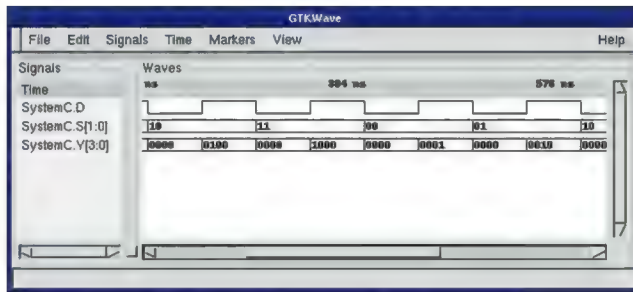
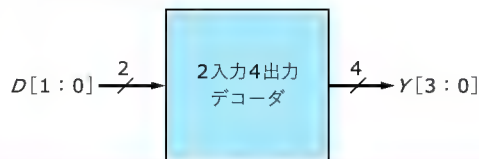



表 6 の真理値表と同じ機能を実現していることが確認できる．

〔図 20〕デマルチプレクサの合成結果

〔図 21〕2 入力 4 出力デコーダの入力と出力

2 入力 4 出力デコーダは，2 ビットの入力線 $D[1:0]$ と，4 ビットの出力線 $Y[3:0]$ をもつ．

〔表 7〕2 入力 4 出力デコーダの真理値表

入力 D		入力 D の	出力 Y			
D_1	D_0	10 進数の値	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	1
0	1	1	0	0	1	0
1	0	2	0	1	0	0
1	1	3	1	0	0	0

2 入力 4 出力デコーダでは，入力 $D[1:0]$ の 10 進数としての値が i であるとき，出力線 $Y[i]$ から 1 を出力し，他の出力線からは，すべて 0 を出力する．

〔リスト 14〕デコーダのヘッダファイルの記述例（decoder.h）

```
#include "systemc.h"

SC_MODULE(decoder) {
  sc_in<sc_uint<2> >  D;
  sc_out<sc_uint<4> > Y;

  void decoder_rtl(void) ;

  SC_CTOR(decoder) {
    SC_METHOD(decoder_rtl) ;
    sensitive << D;
  }
};
```

〔リスト 15〕デコーダのインプリメンテーションファイルの記述例（decoder.cpp）

```
#include "decoder.h"

void decoder::decoder_rtl(void) {
  sc_uint<4> TMP_Y;

  TMP_Y = 0x0;

  switch (D.read())  {
    case 0x0 : TMP_Y[0] = 0x1; break;
    case 0x1 : TMP_Y[1] = 0x1; break;
    case 0x2 : TMP_Y[2] = 0x1; break;
    default  : TMP_Y[3] = 0x1; break;
  }

  Y.write(TMP_Y) ;
};
```

〔リスト 16〕デコーダのシステムファイルの記述例（main_decoder.cpp）

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "decoder.h"

int sc_main(int argc, char *argv[]) {
  int i;
  sc_signal<sc_uint<2> > D;
  sc_signal<sc_uint<4> > Y;

  sc_uint<2> TMP_D;

  decoder DECODER("decoder") ;

  DECODER.D(D) ;
  DECODER.Y(Y) ;

  sc_trace_file *trace_f;
  trace_f = sc_create_vcd_trace_file("decoder_trace") ;
  ((vcd_trace_file *) trace_f) ->sc_set_vcd_time_unit(-9) ;

  sc_trace(trace_f,D,"D") ;
  sc_trace(trace_f,Y,"Y") ;

  cout << "D | Y " << endl;
  cout << "----- " << endl;
  TMP_D = 0x0;
  for(i=0;i<20;i++) {
    if(i>0)  TMP_D++;
    D = TMP_D;
    sc_start(50,SC_NS) ;
    cout <<  D <<" | "<< Y << endl;
  }

  sc_close_vcd_trace_file(trace_f) ;

  return 0;
}
```

信号 D には，50[ns] ごとに 00，01，10，11 が順番に入力され，これが繰り返される．

〔図22〕デコーダのシミュレーション波形

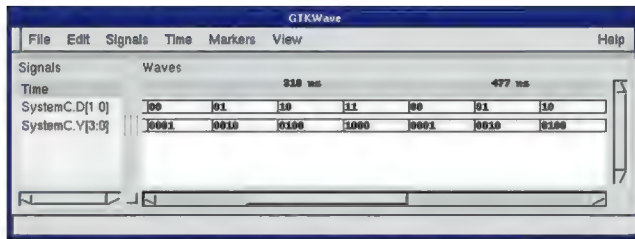



表7の真理値表と同じ機能を実現していることが確認できる．

〔図23〕デコーダの合成結果

〔図24〕4入力2出力エンコーダの入力と出力

4入力2出力エンコーダは，4ビットの入力線D[3：0]と，2ビットの出力線Y[1：0]をもつ．

〔表8〕4入力2出力エンコーダの真理値表

入力D				出力Yの10進数の値	出力Y	
D_3	D_2	D_1	D_0		Y_1	Y_0
0	0	0	1	0	0	0
0	0	1	0	1	0	1
0	1	0	0	2	1	0
1	0	0	0	3	1	1

4入力2出力エンコーダでは，4ビットの入力D[3：0]のうち1本のみが1で残りが0となるような入力を想定している．入力$D[i]$が1で残りが0である場合，出力線Y[1：0]の10進数のしての値がiとなるように出力する．

数で出力する組み合わせ回路をエンコーダ(encoder)と呼びます．より一般的には，デコーダの場合とは逆に，入力信号を何らかの符号に変換する回路を総称してエンコーダと呼びます．

たとえば，4ビットの入力をもつ4入力2出力エンコーダの外観は**図24**のようになり，その真理値表は，**表8**のようになります．

4入力2出力エンコーダのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト17**，**リスト18**に示します．

リスト18に示したインプリメンテーションファイルでは，デコーダの場合と同様に，case文を用いて，信号線Dの値で場合分けをして，エンコーダの出力を決定しています．

ところで，4入力2出力エンコーダでは，4本の入力があるので，入力の組み合わせは全部で16(=2^4)通りあることになります．この16通りの入力のうち，1本の入力線のみ1で，残りが0となるような入力を，エンコーダでは想定しています．これ以外の入力がこのエンコーダに入力されると，**リスト18**からもわかるように，00が出力されます．今回は，想定していない入力に対して00を出力するようにしましたが，回路の用途に応じて，この処理を変更することもできます．

なお，エンコーダのシステムファイルの記述例を**リスト19**に，シミュレーション結果を**図25**に示します．**図25**から，エンコーダとして動作していることが確認できます．また，**リスト17**，**リスト18**をVerilog-HDL記述に変換し，それを論理合成した結果を**図26**に示します．

● コンパレータの設計

二つの2進数の(10進数としての)値の大きさを比較する組み合わせ回路をコンパレータ(comparator)または比較器と呼びます．

大小比較の結果を出力するような回路は，すべてコンパレータと呼ばれます．そこで，ここでは，4ビットのデータA，Bを

〔リスト17〕エンコーダのヘッダファイルの記述例(encoder.h)

```
#include "systemc.h"

SC_MODULE(encoder) {
  sc_in<sc_uint<4> >  D;
  sc_out<sc_uint<2> > Y;

  void encoder_rtl(void) ;

  SC_CTOR(encoder) {
    SC_METHOD(encoder_rtl) ;
    sensitive << D;
  }
};
```

〔リスト18〕エンコーダのインプリメンテーションファイルの記述例(encoder.cpp)

```
#include "encoder.h"

void encoder::encoder_rtl(void) {
  sc_uint<2> TMP_Y;

  switch (D.read())  {
    case 0x8 : TMP_Y = 0x3; break;
    case 0x4 : TMP_Y = 0x2; break;
    case 0x2 : TMP_Y = 0x1; break;
    default  : TMP_Y = 0x0; break;
  }

  Y.write(TMP_Y) ;
};
```

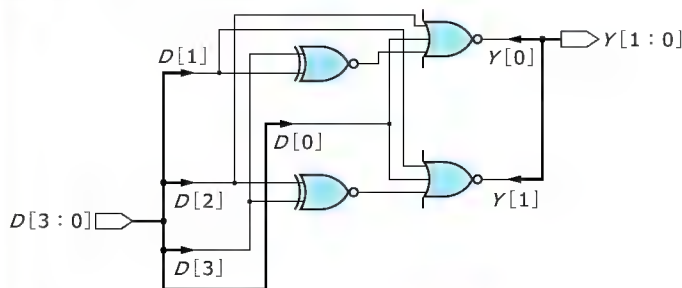
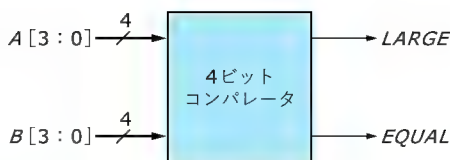
入力とし，この2数の関係が$A=B$，$A>B$，$A<B$のいずれであるかを出力するような回路を設計します．たとえば，この回路の出力をLARGE，EQUALの2本とします．ここで，出力LARGEには，$A>B$のときに1，そうでないときに0を出力させ，また，出力EQUALには，$A=B$のときに1，そうでないときに0を出力させることにします．すなわち，今回設計するコンパレータの外観は，**図27**のようになります．

今回設計するコンパレータの入力は，合計8本あります．すべての入力の組み合わせは，256(=2^8)通りあるので，そのまま真理値表を書くと256行もある表になってしまいます．そこで，

〔リスト 19〕エンコーダのシステムファイルの記述例（main_encoder.cpp）

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "encoder.h"

int sc_main(int argc, char *argv[]) {
  int i, j;
  sc_signal<sc_uint<4> > D;
  sc_signal<sc_uint<2> > Y;

  sc_uint<4> TMP_D;

  encoder ENCODER("encoder") ;

  ENCODER.D(D) ;
  ENCODER.Y(Y) ;

  sc_trace_file *trace_f;
  trace_f = sc_create_vcd_trace_file("encoder_trace") ;

  ((vcd_trace_file *) trace_f) ->sc_set_vcd_time_unit(-9) ;

  sc_trace(trace_f,D,"D") ;
  sc_trace(trace_f,Y,"Y") ;

  cout << "D | Y " << endl;
  cout << "------" << endl;
  for(i=0;i<5;i++) {
    for(j=0;j<4;j++) {
      TMP_D = 0x0;
      TMP_D[j] = 0x1;
      D = TMP_D;
      sc_start(50,SC_NS) ;
      cout <<  D <<" | "<< Y << endl;
    }
  }

  sc_close_vcd_trace_file(trace_f) ;

  return 0;
}
```

信号 D には，50[ns] ごとに 0001，0010，0100，1000 が順番に入力され，これが繰り返される．

〔図 26〕エンコーダの合成結果

〔図 27〕4ビットコンパレータの入力と出力

〔表 9〕コンパレータの入出力対応表

入力 A, B の関係	出力 LARGE	EQUAL
$A = B$	0	1
$A > B$	1	0
$A < B$	0	0

ここで設計するコンパレータは，二つの 4 ビットデータ A[3 : 0]，B[3 : 0] の関係が，$A = B$，$A > B$，$A < B$ のいずれであるかを表すために，2 本の出力 LARGE，EQUAL をもつ．2 本の出力がともに 0 となる場合は，$A < B$ を表していることになる．

〔図 25〕エンコーダのシミュレーション波形

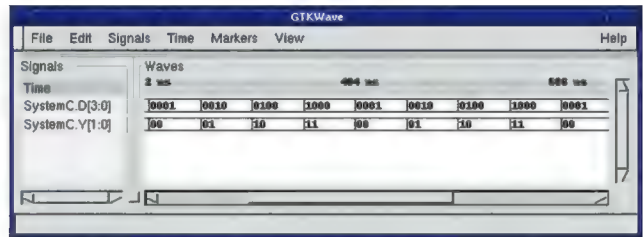



表 8 の真理値表と同じ機能を実現していることが確認できる．

入力 A，B と出力 LARGE，EQUAL との関係を簡単に表すことにします．

このコンパレータにおいて，2 本の出力は，それぞれ $A > B$，$A = B$ であることを表しているため，2 本の出力が同時に 1 になることはありません．すなわち，2 本の出力の一方が 1 で他方が 0 の場合，入力 A，B の関係は，$A > B$，$A = B$ のいずれかになります．残りは，2 本の出力がともに 0 になる場合です．この場合は，$A < B$ であることを表しています．この入力 A，B と，出力 LARGE，EQUAL の対応を表にすると，**表 9** のようになります．

次に，コンパレータのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト 20**，**リスト 21** に示します．**リスト 21** に示したインプリメンテーションファイルでは，if 文を用いて，入力 A，B の関係に応じて，出力 LARGE，EQUAL の値を決定しています．

また，コンパレータのシステムファイルの記述例を**リスト 22** に，シミュレーション結果を**図 28** に示します．**図 28** から，コンパレータとして動作していることが確認できます．さらに，**リスト 20**，**リスト 21** を Verilog-HDL 記述に変換し，それを論理合成した結果を**図 29** (p.74) に示します．

● パリティチェッカの設計

いま，決まった長さの 2 進数列において，その中に現れる 1 の数が奇数または偶数になるようにあらかじめ定めておいたとします．このとき，そのような 2 進数列の 1 ビットに何らかの原因で誤りが生じた場合，1 の数の偶奇を調べることによって，誤りの有無を検査できます．このような検査をパリティチェック (parity check) といいます．

ここで，元の 2 進数列に対して，1 の数が奇数 (偶数) となるように，新たな 1 ビットを付加することによって，パリティチェックを行うことができます．このとき付加した 1 ビットを奇数 (偶数) パリティビット〔odd (even) parity bit〕といい，パリティビットを生成する回路をパリティジェネレータ (parity generator) と呼びます．一方，パリティチェックを行う回路をパリティチェッカ (parity checker) と呼びます．

パリティジェネレータもパリティチェッカも，同じ構成になり，2 進数の各桁の排他的論理和 (XOR) を求めることによって実現できます．ここでは，8 ビットのパリティチェッカを設計

〔リスト 20〕コンパレータのヘッダファイルの記述例(comparator.h)

```
#include "systemc.h"

SC_MODULE(comparator) {
  sc_in<sc_uint<4> >  A;
  sc_in<sc_uint<4> >  B;
  sc_out<bool>        LARGE;
  sc_out<bool>        EQUAL;

  void comparator_rtl(void) ;

  SC_CTOR(comparator) {
    SC_METHOD(comparator_rtl) ;
    sensitive << A << B;
  }
};
```

します．まず，8ビットパリティチェッカの外観と，入出力の対応表を，それぞれ**図 30**，**表 10** に示しておきます．

なお，元の2進数列に奇数パリティビットを付加した場合，パリティチェッカの出力が1のときは正常で，0のときに誤りがあることを表します．一方，元の2進数列に偶数パリティビットを付加した場合，パリティチェッカの出力が0のときは

〔リスト 21〕コンパレータのインプリメンテーションファイルの記述例(comparator.cpp)

```
#include "comparator.h"

void comparator::comparator_rtl(void) {
  sc_uint<4> TMP_A;
  sc_uint<4> TMP_B;
  bool TMP_L;
  bool TMP_E;

  TMP_A = A.read() ;
  TMP_B = B.read() ;

  if ( TMP_A > TMP_B )  {
    TMP_L = true;
    TMP_E = false;
  } else if ( TMP_A == TMP_B )  {
    TMP_L = false;
    TMP_E = true;
  } else {
    TMP_L = false;
    TMP_E = false;
  }

  LARGE.write(TMP_L) ;
  EQUAL.write(TMP_E) ;
};
```

〔リスト 22〕コンパレータのシステムファイルの記述例(main_comparator.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "comparator.h"

int sc_main(int argc, char *argv[]) {
  int i;
  sc_signal<sc_uint<4> > A;
  sc_signal<sc_uint<4> > B;
  sc_signal<bool>        LARGE;
  sc_signal<bool>        EQUAL;

  sc_uint<4> TMP_A;
  sc_uint<4> TMP_B;

  comparator COMPARATOR("comparator") ;

  COMPARATOR.A(A) ;
  COMPARATOR.B(B) ;
  COMPARATOR.LARGE(LARGE) ;
  COMPARATOR.EQUAL(EQUAL) ;

  sc_trace_file *trace_f;
  trace_f = sc_create_vcd_trace_file("comparator_trace") ;
  ((vcd_trace_file *) trace_f) ->sc_set_vcd_time_unit(-9) ;

  sc_trace(trace_f,A,"D") ;
  sc_trace(trace_f,B,"S") ;
  sc_trace(trace_f,LARGE,"LARGE") ;
  sc_trace(trace_f,EQUAL,"EQUAL") ;

  cout << " A B | L E " << endl;
  cout << "-----------" << endl;
  TMP_A = 0x0;
  TMP_B = 0x0;
  for(i=0;i<300;i++) {
    if(i>0)              TMP_A++;
    if(!(i%16) &(i>0))  TMP_B++;
    A = TMP_A;
    B = TMP_B;
    sc_start(50,SC_NS) ;
    cout <<  A <<" " << B <<" | "<< LARGE <<
                              " "<< EQUAL << endl;
  }

  sc_close_vcd_trace_file(trace_f) ;

  return 0;
}
```

〔図 28〕コンパレータのシミュレーション波形

信号 *A* には，50[ns]ごとに 0000，0001，…，1111 が順番に入力され，これが繰り返される．また，信号 *B* には，800[ns]ごとに 0000，0001，…，1111 が順番に入力され，同じく，これが繰り返される．

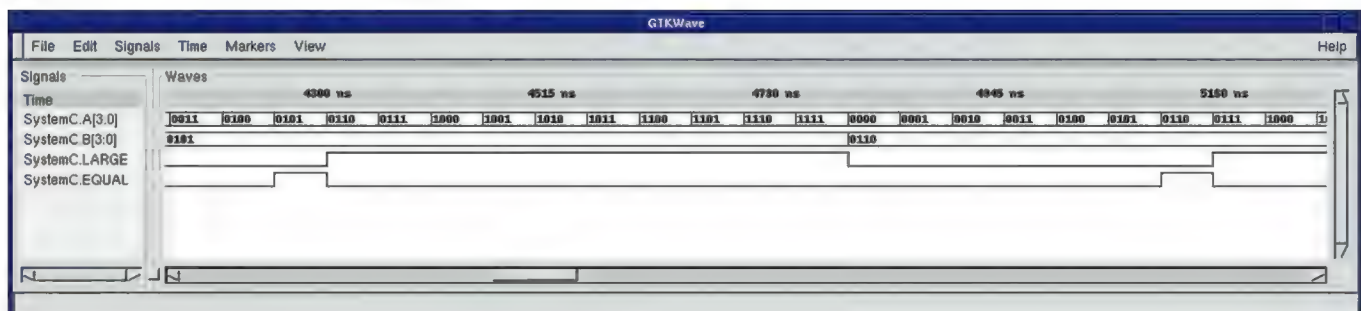
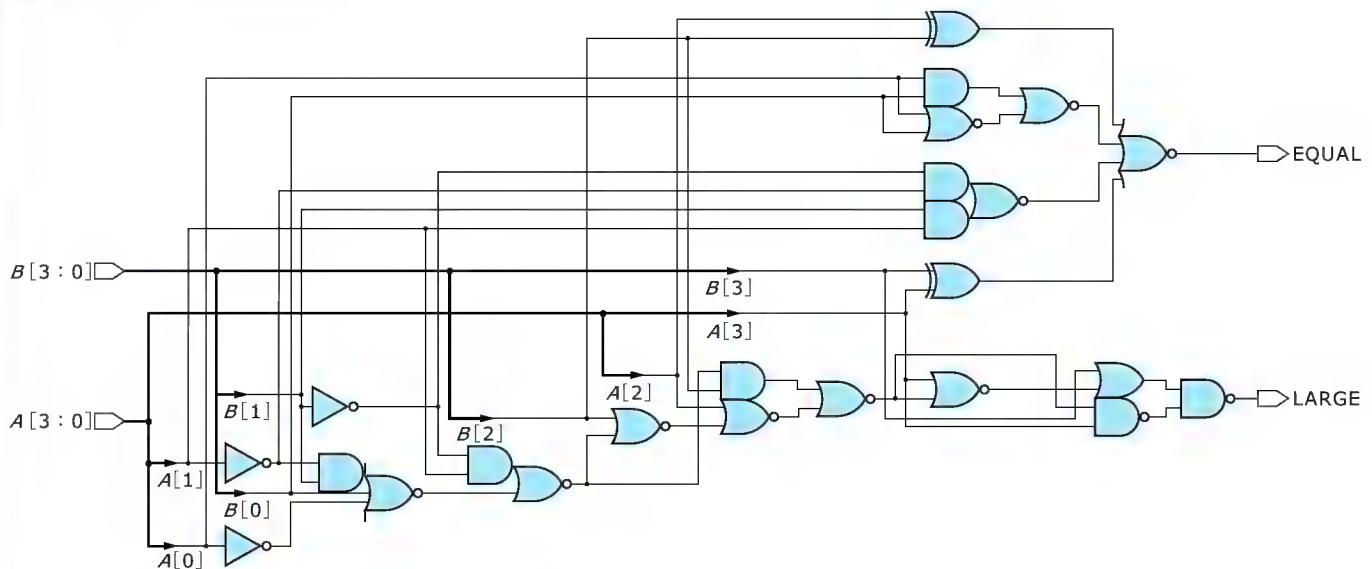
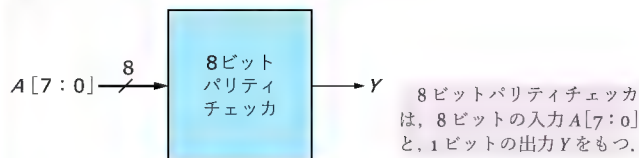



表 9 の対応表に示したとおりの機能を実現していることが確認できる．

〔図29〕コンパレータの合成結果

〔図30〕8ビットパリティチェッカの入力と出力

8ビットパリティチェッカは，8ビットの入力 $A[7:0]$ と，1ビットの出力 Y をもつ．

〔表10〕8ビットパリティチェッカの入出力対応表

入力 A に含まれる1の数	出力 Y
偶数	0
奇数	1

8ビットパリティチェッカでは，入力 $A[7:0]$ の8本の入力のうち，1である入力の数が偶数の場合に0，奇数の場合に1を出力する．

〔リスト25〕パリティチェッカのインプリメンテーションファイルの記述例2(parity_checker.cpp)

```
#include "parity_checker.h"

void parity_checker::parity_checker_rtl(void) {
  bool        TMP_Y;

  TMP_Y = A.read() .xor_reduce() ;

  Y.write(TMP_Y) ;
};
```

.xor_reduce()を用いて，各桁の排他的論理和(XOR)を求めている．

〔リスト23〕パリティチェッカのヘッダファイルの記述例(parity_checker.h)

```
#include "systemc.h"

SC_MODULE(parity_checker) {
  sc_in<sc_uint<8> > A;
  sc_out<bool>       Y;

  void parity_checker_rtl(void) ;

  SC_CTOR(parity_checker) {
    SC_METHOD(parity_checker_rtl) ;
    sensitive << A;
  }
};
```

〔リスト24〕パリティチェッカのインプリメンテーションファイルの記述例1(parity_checker.cpp)

```
#include "parity_checker.h"

void parity_checker::parity_checker_rtl(void) {
  int         i;
  sc_uint<8> TMP_A;
  bool        TMP_Y;

  TMP_A = A.read() ;
  TMP_Y = 0x0;

  for (i=0;i<8;i++) {
    TMP_Y = TMP_Y ^ TMP_A[i];
  }

  Y.write(TMP_Y) ;
};
```

for文を用いて，各桁の排他的論理和(XOR)を求めている．

正常で，1のときに誤りがあることを表します．**表10**では，1の数の偶奇を出力としていますが，その偶奇の意味は，元の2進数列に付加したパリティビットによって異なります．

次に，8ビットパリティチェッカのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト23**，**リスト24**に示します．

リスト24に示したインプリメンテーションファイルでは，for文を用いて，下位桁から順番に排他的論理和(XOR)を求めています．この記述は，じつは**リスト25**に示すように簡単にできます．**リスト25**のように，変数や信号線に続けて.xor_reduce()と記述することによって，その変数や信号線の各桁を順番に排他的論理和(XOR)を取り，最終的に求まった1ビットのデータを返してくれます．SystemCには，同様な機能として，.and_reduce()，.or_reduce()などもありま

〔リスト26〕パリティチェッカのシステムファイルの記述例
(main_parity_checker.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "parity_checker.h"

int sc_main(int argc, char *argv[]) {
  int i;
  sc_signal<sc_uint<8> > A;
  sc_signal<bool>        Y;

  sc_uint<8> TMP_A;

  parity_checker PARITY_CHECKER
                          ("parity_checker");

  PARITY_CHECKER.A(A);
  PARITY_CHECKER.Y(Y);

  sc_trace_file *trace_f;
  trace_f = sc_create_vcd_trace_file("parity_checker_trace");
  ((vcd_trace_file *) trace_f)->sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,A,"A");
  sc_trace(trace_f,Y,"Y");

  cout << "D S | Y " << endl;
  cout << "--------" << endl;
  TMP_A = 0x0;
  for(i=0;i<300;i++) {
    if(i>0)  TMP_A++;
    A = TMP_A;
    sc_start(50,SC_NS);
    cout <<  A <<" | "<< Y << endl;
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

信号Aには，50[ns]ごとに00000000(0x0)，00000001(0x1)，…，11111111(0xFF)が順番に入力され，これが繰り返される．

〔図31〕パリティチェッカのシミュレーション波形

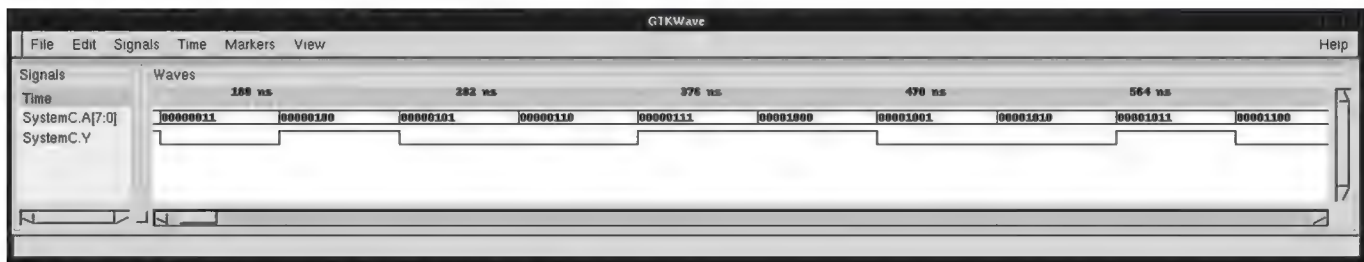



表10に示したとおりの機能を実現していることが確認できる．

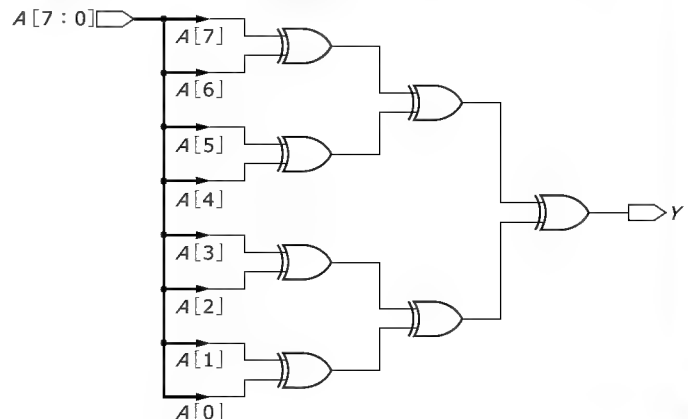
す．それぞれ，.xor_reduce()の論理積(AND)版と論理和(OR)版になります．

なお，パリティチェッカのシステムファイルの記述例を**リスト26**に，パリティチェッカのシミュレーション結果を**図31**に示します．**図31**から，パリティチェッカとして動作していることが確認できます．また，**リスト23**，**リスト24**をVerilog-HDL記述に変換し，それを論理合成した結果を**図32**に示します．

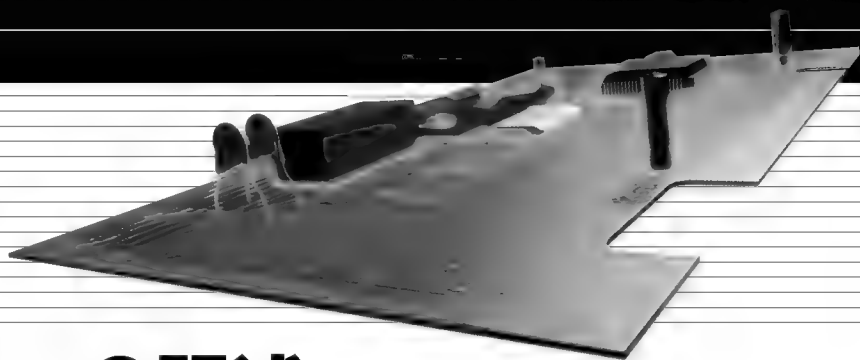
〔図32〕パリティチェッカの合成結果

まとめ

本章では，よく用いられる組み合わせ回路とそのSystemC記述を示してきました．組み合わせ回路は，動作も構造も比較的に単純なので，そのSystemC記述も簡単になります．しかし，SystemC本来の特徴を生かすためには，もっと大きく複雑なシステムの設計に使用しなければなりません．そのような大きく複雑なシステムの記述では，本章で紹介したような記述は，ほとんど現れないでしょう．このあたりのことは，第1章でも説明しました．本特集の後半では，より実際的なSystemCの使用法について解説しますので，そちらもぜひ読んでみてください．

とにかく本章を最後まで読みとおせた方は，ディジタル回路の基礎とSystemCの記述方法をなんとなくわかっていただけたのではないかと思います．しかし，ここはまだ入口です．ディジタル回路もSystemCも，その奥はとても深いので，次章では，もう少し中を覗いてみることにしましょう．

よしだ・たけお　琉球大学 工学部 情報工学

Chapter 3

記憶機能のある
ディジタル回路を記述する

順序回路とSystemC記述

吉田たけお

第2章に続いて，SystemCでディジタル回路設計を行う．この章では，これまでの章で得た知識を元に，ディジタル回路に必須となる順序回路の設計を行う．また応用として，カウンタ，メモリなどをSystemCで記述する．

以上から，本章を読み進めれば，ディジタル回路の基本をSystemCで記述できることになるであろう．

（編集部）

はじめに

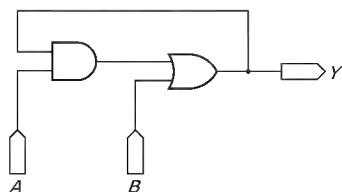
本特集では，C言語に関する知識はあるけれど，ハードウェアについてはあまり詳しくない，という方を対象に，SystemCの基礎について解説しています．第1章では，SystemCの特徴や記述方法などの概要を説明しました．また第2章では，実際の組み合わせ回路をSystemCで記述する方法について見ていきました．この第3章では，組み合わせ回路よりも複雑な順序回路をSystemCで記述する方法について，具体例を示しながら見ていきます．

第2章でも簡単に説明しましたが，順序回路とは，記憶機能をもつディジタル回路です．この記憶機能を実現するもっとも基本的な回路は，フリップフロップと呼ばれます．そこで本章では，まず，フリップフロップの機能や特徴について説明し，フリップフロップおよびそれを応用した順序回路をSystemCで記述する方法や実際の記述例を示していきます．

1　順序回路とは？

● フィードバックのある回路の性質

第2章でもふれましたが，フィードバックのあるディジタル回路は，記憶機能を実現してしまう場合があります．そのようすを，**図1**に示すフィードバックのあるディジタル回路で見てみましょう．

〔図1〕フィードバックのあるディジタル回路の例

まず，**図1**の回路において，入力Aを0として，入力Bを0→1→0→1→0と変化させた場合，出力Yがどのように変化するか見てみましょう．この場合，入力Aの値が0なので，フィードバックの値に関わらず，ANDゲートの出力は0となります．さらに，ANDゲートの出力が0になるので，入力Bの値が，そのままORゲートの出力となります．すなわち，出力Yは，0→1→0→1→0となり，入力Bと同じように変化します．入力が変化すると出力も変化しているので，この場合，記憶機能を実現していることにはなりません．

次に，入力Bを0として，入力Aを1→0→1→0→1と変化させた場合について見てみましょう．この場合，まず入力Aの値が1なので，フィードバックの値が，そのままANDゲートの出力となります．この時点では，フィードバックの値がわからないので，とりあえず“？”としておきます．また，入力Bの値が0なので，ANDゲートの出力が，そのままORゲートの出力となります．すなわち，ORゲートの出力は，？です．この状態で，入力Aの値が0になると，フィードバックの値に関わらず，ANDゲートの出力が0になります．この値0は，そのまま，ORゲートの出力，フィードバックの値，出力Yの値になります．次に，入力Aの値が1になりますが，フィードバックの値が0なので，ANDゲートの出力，ORゲートの出力，出力Yの値はすべて0になります．これ以降，入力Aの値をどのように変化させても，出力Yの値は0のままになります．すなわち出力Yは，？→0→0→0→0のようになります．この場合，入力が変化しても，出力が0のままで変化していないので，0を記憶（保持）していることになります．

同様に，入力Aを1として，入力Bを0→1→0→1→0と変化させると，出力Yは，？→1→1→1→1のように変化します．この場合は，1を記憶（保持）していることになります．

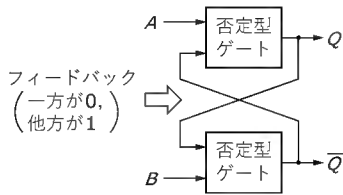
以上で見たように，フィードバックのあるディジタル回路では，入力の与え方によっては，記憶機能を実現できる場合があります．このことを利用して1ビットのデータを保持するディ

〔図2〕フリップフロップの基本構成

〔図3〕RS-FFの記号

FFは，ゲート回路と同様に，ディジタル回路を構成する最小単位の部品である．そのため，回路図の中では，このような記号を用いて表される．

〔図4〕RS-FFの回路構成

RS-FFは，二つのNORゲートを用いて実現できる．

ジタル回路が，フリップフロップです．以下では，フリップフロップについて見てみましょう．

● フリップフロップの基本構成

フリップフロップ（FF）は，**図2**に示すように，NANDゲートやNORゲートなどの否定型のゲートを2個用い，それぞれのゲートの出力をもう一方のゲートの入力とすることによって構成されます．このとき，二つあるフィードバックの一方の値が0になると，もう一方の値が1になるため，二つのゲート回路の出力が保持されます．FFには，いくつかの種類があるので，まず，もっとも基本的なRS-FF（reset-set FF）について説明します．

FFは，ゲート回路と同様に，ディジタル回路を構成する最小単位の部品となります．そのため回路図中では，FF専用の記号を用いて表されます．たとえば，RS-FFの場合，**図3**のような簡単な記号になります．この記号の中身，すなわち，RS-FFの構成は，**図4**のようになっています．

RS-FFには，RとSの2本の入力線があるので，全部で4通りの入力を与えることができます．以下で，それぞれの場合のRS-FFの動作について確認してみましょう．なお以下の説明でQ^tは，時刻tにおけるQの値を表しています．

1）$R=1$，$S=0$の場合

図4から，出力Qは，出力$\overline{Q}$と入力Rの論理和否定（NOR）となっていることがわかります．これに，時刻を考慮すると，

$$Q^{t+1}=\overline{\overline{Q}^{t}+R} \quad \cdots\cdots (1)$$

と表すことができます．ここで，入力Rが1なので，Q^tの値とは無関係に$Q^{t+1}=0$となることがわかります．また，このことから，$\overline{Q}^{t+1}=1$となることもわかります．なお，$Q^{t-1}=0$とすることを，FFをリセット（reset）するといいます．

2）$R=0$，$S=1$の場合

図4から出力$\overline{Q}$は，出力Qと入力Sの論理和否定（NOR）となっていることがわかります．これに，時刻を考慮すると，

$$\overline{Q}^{t+1}=\overline{Q^{t}+S} \quad \cdots\cdots (2)$$

と表すことができます．ここで，入力Sが1なので，Q^tの値とは無関係に$\overline{Q}^{t+1}=0$となることがわかります．また，このことから，$Q^{t+1}=1$となることもわかります．なお，$Q^{t+1}=1$とすることを，FFをセット（set）するといいます．

3）$R=0$，$S=0$の場合

まず，入力Rが0なので，式(1)から，$Q^{t+1}=Q^t$となります．また同様に，入力Sが0なので，式(2)から，$\overline{Q}^{t+1}=\overline{Q}^t$となります．このことは，前時刻の出力が保持（記憶）されていることを表しています．すなわち，RS-FFは，$R=0$，$S=0$の場合，記憶機能を実現します．

4）$R=1$，$S=1$の場合

NORゲートの性質から，$Q^{t+1}=\overline{Q}^{t+1}=0$となります．しかしこの後，$R=S=0$と変化した場合を考えてみましょう．

まず，**図4**の上側のNORゲートを先に見た場合，入力Sが0になると，その出力は1となります．このとき，下側のNORゲートの出力は0のままです．すなわち，$Q^{t+1}=0$となります．これで良さそうですが，今度は，先に下側のNORゲートを見てみます．下側のNORゲートも，入力Rが0になると，その出力は1となります．このとき，上側のNORゲートの出力は0のままとなります．すなわち，$Q^{t+1}=1$となります．このように，RS-FFに，$R=S=1$を入力してから，$R=S=0$と変化させると，$Q^{t+1}=0$，$Q^{t+1}=1$のいずれになるのかがわからなくなってしまいます．そのため，RS-FFでは，$R=S=1$となる入力の組み合わせは禁止されています．

以上のことを表にまとめると，**表1**のようになります．**表1**のようなFFの動作をまとめた表を，特性表（characteristic table）と呼びます．特性表は，ゲート回路や組み合わせ回路の真理値表と同様に，そのFFの機能を表しますが，時間の概念が含まれている点で，真理値表とは異なっています．

なおFFには，RS-FFのほかに，JK-FF，T-FF，D-FFなどがあります．これらのFFは，RS-FFに，いくつかのゲート回路を付け加えることによって実現できますが，その詳細は割愛します．

● 安定動作をするフリップフロップ

ところでRS-FFでは，その二つの入力R，Sが同時に変化することを想定しています．この条件が満たされる場合は，**表1**の特性表に示されたとおりの動作をします．しかし，二つの入力R，Sが同時に変化しなかった場合は，瞬間的に異なる値を

〔表1〕**RS-FFの特性表**

入力		出力	
S	R	Q^{t+1}	$\overline{Q}^{t+1}$
0	0	Q^{t}	$\overline{Q}^{t}$
0	1	0	1
1	0	1	0
1	1	禁止	禁止

特性表は，FFの動作をまとめた表である．RS-FFでは，$R=S=0$のときに記憶機能を実現する．なお，RS-FFでは，$R=S=1$の組み合わせは禁止入力とされている．

〔表2〕**エッジトリガ型D-FFの特性表**

入力		出力	
CLK	D	Q^{t+1}	$\overline{Q}^{t+1}$
↓	*	Q^{t}	$\overline{Q}^{t}$
↑	0	0	1
↑	1	1	0

(**a**) ポジティブエッジトリガ型D-FFの特性表

入力		出力	
CLK	D	Q^{t+1}	$\overline{Q}^{t+1}$
↑	*	Q^{t}	$\overline{Q}^{t}$
↓	0	0	1
↓	1	1	0

(**b**) ネガティブエッジトリガ型D-FFの特性表

この特性表において，↑は，クロックCLKの立ち上がりエッジを，↓は，クロックCLKの立ち下がりエッジを，それぞれ表している．また，*は，ドントケアと呼ばれ，その入力が，0でも1でもよいことを表している．

出力[注1]したり，まったく異なる動作をする可能性があります．このような問題を解決するために，入力を取り込むタイミングを指定するための信号を設けたFFが，実際には良く用いられます．この形式のFFは，エッジトリガ型FF(edge-triggered FF)と呼ばれます．

エッジトリガ型FFでは，クロック(clock)信号と呼ばれる信号が変化した瞬間に，入力を取り込みます．ここで，信号が，0から1に変化する瞬間を立ち上がりエッジ(positive edge)，1から0に変化する瞬間を立ち下がりエッジ(negative edge)といいます．このとき，クロック信号の立ち上がりエッジで入力を取り込むFFを，ポジティブエッジトリガ型FF(positive-edge-triggered FF)，クロック信号の立ち下がりエッジで入力を取り込むFFを，ネガティブエッジトリガ型FF(negative-edge-triggered FF)と呼びます．

ところで，構造的にもっとも単純なFFはRS-FFですが，機能的にもっとも単純なFFは，D-FFと呼ばれるFFです．実際のディジタル回路設計では，このD-FFをエッジトリガ型にしたエッジトリガ型D-FFがよく用いられます．そこで，このエッジトリガ型D-FFの回路記号を**図5**に，その特性表を**表2**に，それぞれ示しておきます．

図5に示すように，エッジトリガ型FFの記号には，クロック信号に>印を付けます．また，ネガティブエッジトリガ型FFの場合，さらに，○印も付けます．なお，**表2**において，↑は，クロックCLKの立ち上がりエッジを，↓は，クロックCLKの立ち下がりエッジを，それぞれ表しています．また，*は，ドントケア(don t care)と呼ばれ，その入力が，0でも1でもよいことを表しています．

表2に示すように，エッジトリガ型D-FFは，クロック信号の立ち上がりエッジ(立ち下がりエッジ)で，データ入力Dを取り込み，それをそのまま，出力Qの値とする単純な機能をもったFFです．

● 順序回路の構成

先にも述べましたが，順序回路は，記憶機能をもったディジタル回路です．この記憶機能を実現する回路がFFです．すなわち，FFは，もっとも簡単な順序回路ということができます．順序回路は，このFFと各種ゲート回路を用いて構成されます．このような順序回路は，ゲート回路やFFなどの各部品の出力を，他の部品の入力に接続する，という手順を繰り返すことによって構成されます．

ここで，個々のFFは，単純な動作を実現しますが，それが多数組み合わされると，全体としては，非常に複雑な動作を実現できます．そのため，単純に上記の手順を繰り返すだけでは，所望の回路を設計することは困難です．すなわち，順序回路の設計に際しては，定式化された設計法が重要となります．定式化された設計法に関しては，次の第4章でその概要を簡単に説明します．以下では，まず，いくつかのFFとゲート回路を用いて，比較的簡単に設計できる順序回路を紹介します．

〔図5〕**エッジトリガ型D-FFの記号**

(**a**) ポジティブエッジトリガ型D-FFの記号
(**b**) ネガティブエッジトリガ型D-FFの記号

注1：このように，瞬間的に出力される不正な信号をハザード(hazard)と呼ぶ．

2 順序回路の設計例

● エッジトリガ型Dフリップフロップの設計

まず，もっとも簡単な順序回路であるFFをSystemCで設計してみましょう．ここでは，先に述べたように，ディジタル回路設計でよく用いられるエッジトリガ型D-FFを設計します．なお，シミュレーションなどの設計環境は第2章と同じなので，そちらを参照してください．

エッジトリガ型D-FFについては，説明済みなので，さっそく，そのヘッダファイルとインプリメンテーションファイルを示します．それぞれ，**リスト1**，**リスト2**のとおりです．なお，**リスト1**，**リスト2**は，ポジティブエッジトリガ型D-FFの記

〔リスト 1〕ポジティブエッジトリガ型 D-FF のヘッダファイルの記述例(d_ff.h)

```
#include "systemc.h"

SC_MODULE(d_ff){
  sc_in_clk     CLK;
  sc_in<bool>   D;
  sc_out<bool>  Q;

  void d_ff_behavior(void);

  SC_CTOR(d_ff){
    SC_METHOD(d_ff_behavior);
    sensitive << CLK.pos();
  }
};
```

信号の立ち上がりエッジは，.pos()を用いて表す．立ち下がりエッジの場合は，.neg()とすればよい．

述になっています．また D-FF の反転出力$\overline{Q}$は，省略してあります．

まず，**リスト 1** のヘッダファイルを見てみましょう．ポジティブエッジトリガ型 D-FF では，クロック信号 CLK の立ち上がりエッジで，データ入力 D を取り込み，出力 Q の値とします．この立ち上がりエッジを表す記述が，.pos()になります．すなわち CLK.pos()で，クロック信号 CLK の立ち上がりエッジを表します．これを，センシティビティリストに指定しておけば，クロック信号 CLK の立ち上がりのたびに，プロセス d_ff_behavior が起動されます．なお，立ち下がりエッジを表す場合は，.neg()を用います．

ところで，組み合わせ回路の場合，プロセスに対するどの入力が変化しても，そのプロセスの出力が変化する可能性がありました．そのため，センシティビティリストに，プロセスに対する入力をすべて記述する必要がありました．しかし，**リスト 1** では，クロック信号 CLK のみが記述されており，データ入力 D は，記述されていません．これは，ポジティブエッジトリガ型 D-FF では，データ入力 D が変化しても，クロック信号 CLK が変化しなければ，出力 Q の値が変化しないためです．

このように，クロック信号をともなう回路では，クロック信号の値が変化しなければ，出力信号の値は変化しません．そのため，センシティビティリストには，クロック信号およびクロック信号より優先される信号のみを指定します．この点が組み合わせ回路の記述方法と異なります．なお，クロック信号より優先される信号については，次のメモリレジスタの設計で説明します．

一方，**リスト 2** のインプリメンテーションファイルでは，プロセスが起動されたときに，データ入力 D を出力 Q の値とすることを表しています．

次に，ポジティブエッジトリガ型 D-FF のシステムファイルを**リスト 3** に，シミュレーション結果を**図 6** に示します．**図 6** から，**表 2(a)** の特性表と同じ機能を実現していることがわかります．

〔リスト 2〕ポジティブエッジトリガ型 D-FF のインプリメンテーションファイルの記述例(d_ff.cpp)

```
#include "d_ff.h"

void d_ff::d_ff_behavior(void)
{
  Q.write(D.read());
};
```

〔リスト 3〕ポジティブエッジトリガ型 D-FF のシステムファイルの記述例(main_d_ff.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "d_ff.h"

int sc_main(int argc, char *argv[])
{
  int i;
  sc_clock CLK("CLK",100,SC_NS,0.5,0,SC_NS,false);
  sc_signal<bool>  D;
  sc_signal<bool>  Q;

  d_ff D_FF("d_ff");

  D_FF.CLK(CLK);
  D_FF.D(D);
  D_FF.Q(Q);

  sc_trace_file *trace_f;

  trace_f = sc_create_vcd_trace_file("d_ff_trace");

  ((vcd_trace_file *)trace_f)-> sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,CLK,"CLK");
  sc_trace(trace_f,D,"D");
  sc_trace(trace_f,Q,"Q");

  for (i=0; i<5; i++) {
    D = false;
    sc_start(70,SC_NS);
    D = true;
    sc_start(20,SC_NS);
    D = false;
    sc_start(20,SC_NS);
    D = true;
    sc_start(80,SC_NS);
    D = false;
    sc_start(30,SC_NS);
    D = true;
    sc_start(80,SC_NS);
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

また，**リスト 1** のヘッダファイルと**リスト 2** のインプリメンテーションファイルを Verilog-HDL 記述に変換し，それを論理合成した結果を**図 7** に示します．**図 7** は，**図 5(a)** に示した記号と同じような図になっています．このことからも，論理合成ツールにおいて，ポジティブエッジトリガ型 D-FF が，ディジタル回路の部品として扱われていることがわかります．

なお以降では，このエッジトリガ型 D-FF のことを単に D-FF または FF と呼ぶことにします．

● メモリレジスタの設計

先に述べたように，FF は，1 ビットのデータを記憶すること

が可能な順序回路です．この性質を利用して，**図8**のように，n個のFFを用いてnビットの2進数データを記憶する回路を構成できます．このような回路を，メモリレジスタ(memory register)あるいは単にレジスタ(register)と呼びます．

ところで，FF内部のフィードバックの値は，そのFFの使用を開始した直後には，0になるか1になるかがわかりません．このことが原因で，FFを含む回路が誤動作を起こす危険性があります．そのためFFには，そのFFを強制的にリセットする(出力Qを0にする)ためのリセット信号や，強制的にセットする(出力Qを1にする)ためのプリセット信号をもったFFもあり，実際によく用いられています．このとき，クロック信号とは無関係にそのFFをリセットできる信号を非同期リセットと呼びます．また，クロック信号の立ち上がりまたは立ち下がりエッジで，そのFFをリセットできる信号を同期リセットと呼びます．プリセット信号についても同様です．以降では，非同期リセット信号をもったFFを使用することにします．

次に，4ビットメモリレジスタのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト4**，**リスト5**に示します．

非同期リセット信号は，クロック信号とは無関係にそのFFをリセットする信号であり，クロック信号よりも優先されます．このように，クロック信号よりも優先される信号がある場合は，先に述べたように，その信号もセンシティビティリストに指定します．**リスト4**のヘッダファイルでは，リセット信号RESETをセンシティビティリストに指定しています．

なお，**リスト4**のセンシティビティリストでは，リセット信号の指定をRESET.pos()としています．この.pos()は，不必要なように思えますが，.pos()がない場合，クロック信号CLKが変化していなくても，リセット信号RESETの立ち下がりエッジでプロセスが起動されてしまいます．そのため，RESET.pos()としています．なお，この例では，リセット信号RESETが1になったときに，FFがリセットされます．リセット信号RESETが0になったときに，FFをリセットさせる場合は，センシティビティリストにRESET.neg()と指定します．

一方，**リスト5**のインプリメンテーションファイルでは，クロック信号よりも優先されるリセット動作を先に記述しています．この点を除いては，FFの記述と同様になっています．

続いて，4ビットメモリレジスタのシステムファイルを**リスト6**に，シミュレーション結果を**図9**に示します．**図9**から，正しく動作していることが確認できます．

〔図6〕ポジティブエッジトリガ型D-FFのシミュレーション波形

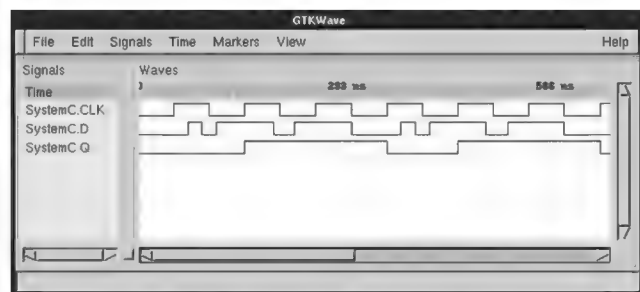
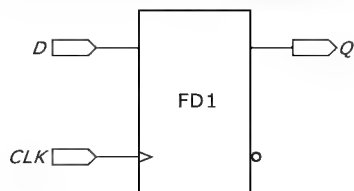
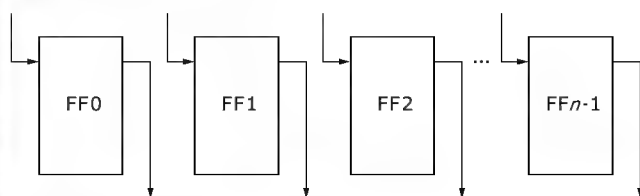



表2(a)の特性表と同じ機能を実現していることが確認できる．

〔図7〕ポジティブエッジトリガ型D-FFの合成結果

〔図8〕nビットメモリレジスタ

〔リスト4〕4ビットメモリレジスタのヘッダファイルの記述例(memory_register.h)

```
#include "systemc.h"

SC_MODULE(memory_register){
  sc_in_clk            CLK;
  sc_in<bool>          RESET;
  sc_in<sc_uint<4> >   D;
  sc_out<sc_uint<4> >  Q;

  void memory_register_behavior(void);

  SC_CTOR(memory_register){
    SC_METHOD(memory_register_behavior);
    sensitive << RESET.pos() << CLK.pos();
  }
};
```

リセット信号やプリセット信号などの，クロック信号よりも優先される信号がある場合は，その信号もセンシティビティリストに指定する必要がある．また，その信号が1になったときに動作する場合は.pos()を，0になったときに動作する場合は.neg()を付ける．

〔リスト5〕4ビットメモリレジスタのインプリメンテーションファイルの記述例(memory_register.cpp)

```
#include "memory_register.h"

void memory_register::memory_register_behavior(void)
{
  if ( RESET.read() == true ) {
    Q.write(0x0);
  } else {
    Q.write(D.read());
  }
};
```

また，**リスト4**のヘッダファイルと**リスト5**のインプリメンテーションファイルをVerilog-HDL記述に変換し，それを論理合成した結果を**図10**に示します．

● シフトレジスタの設計

nビットメモリレジスタのように，nビットのデータを同時に並列に取り込むことを，パラレル(parallel)入力といいます．nビットメモリレジスタでは，出力も各ビット同時に並列に行っているので，パラレル入力，パラレル出力のレジスタになります．これに対して，データを1ビットずつ入力，出力することを，それぞれシリアル(serial)入力，シリアル出力といいます．

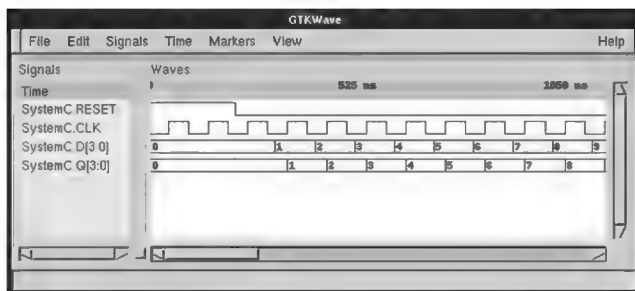
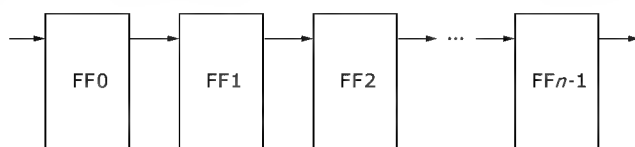
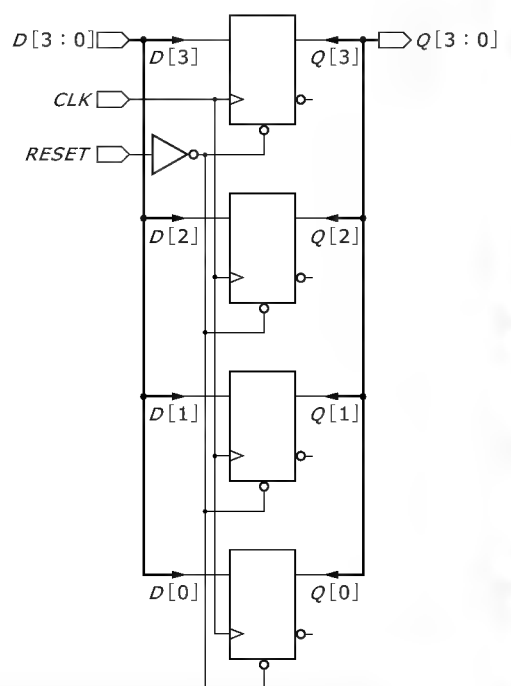
ここで，**図11**に示すように，n個のFFを縦続接続したシリアル入力のレジスタをシフトレジスタ(shift register)と呼びます．

シフトレジスタは，レジスタに保持しているデータを，1ビットずつ移動(シフト)する機能をもっています．また，シリアルデータ(直列データ)をパラレルデータ(並列データ)に変換するシリアル-パラレル変換や，逆にパラレルデータをシリアルデータに変換するパラレル-シリアル変換などにも用いられます．ここでは，シリアル入力，パラレル出力の4ビットシフトレジスタを設計します．

まず，4ビットシフトレジスタのヘッダファイルとインプリ

〔**リスト6**〕**4ビットメモリレジスタのシステムファイルの記述例**(main_memory_register.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "memory_register.h"

int sc_main(int argc, char *argv[])
{
  int i;
  sc_clock CLK("CLK",100,SC_NS,0.5,0,SC_NS,false);
  sc_signal<bool>        RESET;
  sc_signal<sc_uint<4> > D;
  sc_signal<sc_uint<4> > Q;

  sc_uint<4> TMP_D;

  memory_register MEMORY_REGISTER("memory_register");

  MEMORY_REGISTER.CLK(CLK);
  MEMORY_REGISTER.RESET(RESET);
  MEMORY_REGISTER.D(D);
  MEMORY_REGISTER.Q(Q);

  sc_trace_file *trace_f;

  trace_f = sc_create_vcd_trace_file("memory_register_trace");

  ((vcd_trace_file *)trace_f)-> sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,CLK,"CLK");
  sc_trace(trace_f,RESET,"RESET");
  sc_trace(trace_f,D,"D");
  sc_trace(trace_f,Q,"Q");

  RESET = true;
  TMP_D = 0x0;
  sc_start(220,SC_NS);
  RESET = false;
  for (i=0; i<50; i++) {
    D = TMP_D++;
    sc_start(100,SC_NS);
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

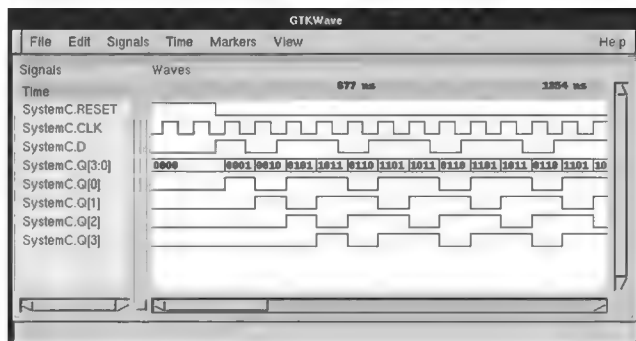
〔**図9**〕**4ビットメモリレジスタのシミュレーション波形**

クロック信号CLKの各立ち上がりエッジにおいて，データ入力Dの値が，FFに取り込まれ，出力Qの値となっていることがわかる．

〔**図10**〕**4ビットメモリレジスタの合成結果**

〔**図11**〕***n*ビットシフトレジスタ**

〔リスト7〕4ビットシフトレジスタのヘッダファイルの記述例(shift_register.h)

```
#include "systemc.h"

SC_MODULE(shift_register){
  sc_in_clk          CLK;
  sc_in<bool>        RESET;
  sc_in<bool>        D;
  sc_out<sc_uint<4> > Q;

  void shift_register_behavior(void);

  SC_CTOR(shift_register){
    SC_METHOD(shift_register_behavior);
    sensitive << RESET.pos() << CLK.pos();
  }
};
```

〔リスト8〕4ビットシフトレジスタのインプリメンテーションファイルの記述例(shift_register.cpp)

```
#include "shift_register.h"

void shift_register::shift_register_behavior(void)
{
  if ( RESET.read() ) {
    Q = 0x0;
  } else {
    Q = ( Q.read().range(2,0), D.read() );
  }
};
```

〔図12〕4ビットシフトレジスタのシミュレーション波形

クロック信号CLKの各立ち上がりエッジにおいて，出力Qの値が1ビット分シフトしていることがわかる．なお，データ入力Dに接続されているFFでは，クロック信号CLKの各立ち上がりエッジにおいて，データ入力Dの値を取り込んでいる．

〔リスト9〕4ビットシフトレジスタのシステムファイルの記述例(main_shift_register.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "shift_register.h"

int sc_main(int argc, char *argv[])
{
  int i;
  sc_clock CLK("CLK",100,SC_NS,0.5,0,SC_NS,false);
  sc_signal<bool>       RESET;
  sc_signal<bool>       D;
  sc_signal<sc_uint<4> > Q;

  shift_register SHIFT_REGISTER("shift_register");

  SHIFT_REGISTER.CLK(CLK);
  SHIFT_REGISTER.RESET(RESET);
  SHIFT_REGISTER.D(D);
  SHIFT_REGISTER.Q(Q);

  sc_trace_file *trace_f;

  trace_f =
sc_create_vcd_trace_file("shift_register_trace");

  ((vcd_trace_file *)trace_f)-> sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,CLK,"CLK");
  sc_trace(trace_f,RESET,"RESET");
  sc_trace(trace_f,D,"D");
  sc_trace(trace_f,Q,"Q");

  RESET = true;
  D = false;
  sc_start(220,SC_NS);
  RESET = false;
  for (i=0; i<20; i++) {
    D = true;
    sc_start(100,SC_NS);
    D = false;
    sc_start(100,SC_NS);
    D = true;
    sc_start(100,SC_NS);
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

メンテーションファイルを，それぞれ**リスト7**，**リスト8**に示します．

リスト8のインプリメンテーションファイルでは，シフト機能を実現するために，連接演算子(,)と信号の範囲指定をする.range()を使用しています．たとえば，3ビットの信号A，Bと4ビットの信号Cを用いて，(A,B,C)と記述した場合，これはA，B，Cをこの順に連結した10ビットの信号を表します．また，B.range(2,0)と記述した場合，4ビットの信号BからB[2]，B[1]，B[0]を取り出した3ビットの信号を表します．

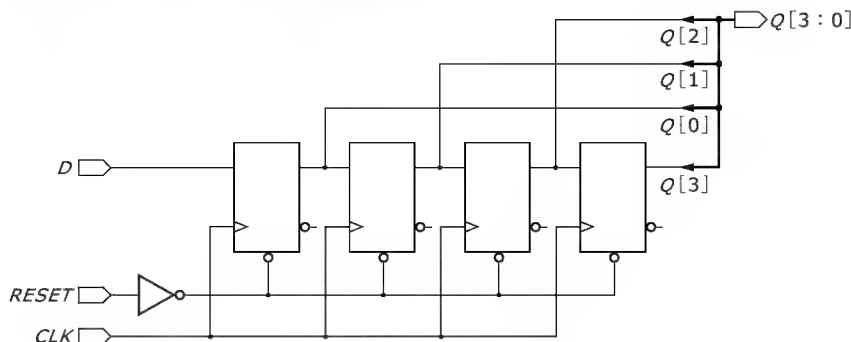
リスト8では，出力Qの下位3ビットと入力Dを連接した4ビットの信号を新たにQに代入することで，シフト機能を表現しています．

次に，4ビットシフトレジスタのシステムファイルを**リスト9**に，シミュレーション結果を**図12**に示します．**図12**から，正しくシフト動作をしていることが確認できます．

また，**リスト7**のヘッダファイルと**リスト8**のインプリメンテーションファイルをVerilog-HDL記述に変換し，それを論理合成した結果を**図13**に示します．

● *N*進カウンタの設計

入力されたクロック信号の立ち上がりエッジあるいは立ち下がりエッジの回数を数える回路をカウンタ(counter)と呼びます．とくに，0から$N-1$までの数を繰り返し数えるカウンタを*N*進カウンタ(modulo-*N* counter)と呼びます．ここでは，パラメータを使って，任意の正整数*N*に対して*N*進カウンタを実現

〔図13〕4ビットシフトレジスタの合成結果

〔リスト10〕*N*進カウンタのヘッダファイルの記述例(counter_n.h)

```
#include "systemc.h"

#define F  4
#define N 10

SC_MODULE(counter_n){
  sc_in_clk           CLK;
  sc_in<bool>         RESET;
  sc_out<sc_uint<F> > Y;

  void counter_n_behavior(void);

  SC_CTOR(counter_n){
    SC_METHOD(counter_n_behavior);
    sensitive << RESET.pos() << CLK.pos();
  }
};
```

〔リスト11〕*N*進カウンタのインプリメンテーションファイルの記述例(counter_n.cpp)

```
#include "counter_n.h"

void counter_n::counter_n_behavior(void)
{
  if ( RESET.read() ) {
    Y = 0x0;
  } else if ( Y.read() == N-1 ) {
    Y = 0x0;
  } else {
    Y = Y.read() + 0x1;
  }
};
```

〔リスト12〕*N*進カウンタのシステムファイルの記述例(main_counter_n.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "counter_n.h"

int sc_main(int argc, char *argv[])
{
  sc_clock CLK("CLK",100,SC_NS,0.5,0,SC_NS,false);
  sc_signal<bool>        RESET;
  sc_signal<sc_uint<F> > Y;

  counter_n COUNTER_N("counter_n");

  COUNTER_N.CLK(CLK);
  COUNTER_N.RESET(RESET);
  COUNTER_N.Y(Y);

  sc_trace_file *trace_f;

  trace_f = sc_create_vcd_trace_file("counter_n_trace");

  ((vcd_trace_file *)trace_f)-> sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,CLK,"CLK");
  sc_trace(trace_f,RESET,"RESET");
  sc_trace(trace_f,Y,"Y");

  RESET = true;
  sc_start(220,SC_NS);
  RESET = false;
  sc_start(5000,SC_NS);

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

できるSystemC記述を示します.

まず，*N*進カウンタのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト10**，**リスト11**に示します.

ところで，*m*個のFFを用いると，最大2^m種類のデータを保持することが可能です．また，*N*進カウンタでは，0から$N-1$までの*N*種類の数を保持できる必要があります．すなわち，*N*カウンタを実現するためには，少なくとも$\log_2 N$個(少数点以下は切り上げ)のFFが必要になります．**リスト10**のヘッダファイルでは，#defineを用いて，FFの数Fと進数Nを変更できるようにしています．**リスト10**では，Fを4，Nを10としているので，FFを4個用いて，0から9までの数を繰り返し数える10進カウンタのヘッダファイルになっています．なお，FとNは，$2^F \geqq N$となるように指定する必要があります.

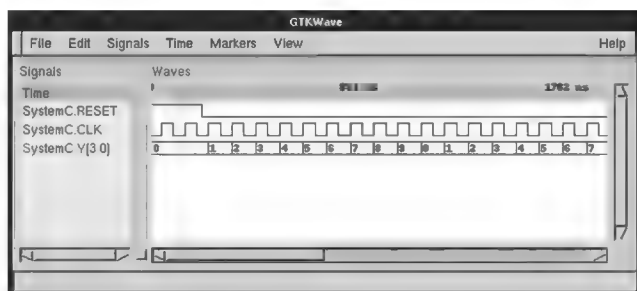
一方，**リスト11**のインプリメンテーションファイルでは，クロック信号の立ち上がりエッジによって，プロセスが起動されるたびに，出力Yの値を1だけ増やしています．このとき，もし，出力Yの値がN-1であれば，再び0から数え直します.

次に，*N*進カウンタのシステムファイルを**リスト12**に，10進カウンタのシミュレーション結果を**図14**に示します．**図14**から，0から9までの数が，繰り返し数えられていることがわかります.

また，**リスト10**のヘッダファイルと**リスト11**のインプリメンテーションファイルをVerilog-HDL記述に変換し，それを論理合成した結果を**図15**に示します.

● アップダウンカウンタの設計

カウンタには，いろいろな種類があります．通常のカウンタは，クロック信号の立ち上がりまたは立ち下がりエッジの回数をカウントアップしていくため，アップカウンタ(up counter)とも呼ばれます．これに対して，カウントダウンしていくカウンタをダウンカウンタ(down counter)と呼びます．また，制御信号により，カウントアップとカウントダウンを切替え可能なカウンタもあり，このようなカウンタをアップダウンカウン

〔図 14〕 **10 進カウンタのシミュレーション波形**

クロック信号 CLK の立ち上がりエッジのたびに，出力 Q の値が 1 だけ増えて，0～9 を繰り返し出力していることがわかる．

タ（up/down counter）と呼びます．ここでは，2^N 進アップダウンカウンタを設計します．

2^N 進アップダウンカウンタのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト 13**，**リスト 14** に示します．

リスト 13 のヘッダファイルでは，`#define` を用いて，FF の数 N を変更できるようにしています．**リスト 13** では，N を 4 としているので，16 進アップダウンカウンタの記述になっています．なお，先に示した N 進カウンタのように記述すれば，N 進アップダウンカウンタの記述に変更することもできます．

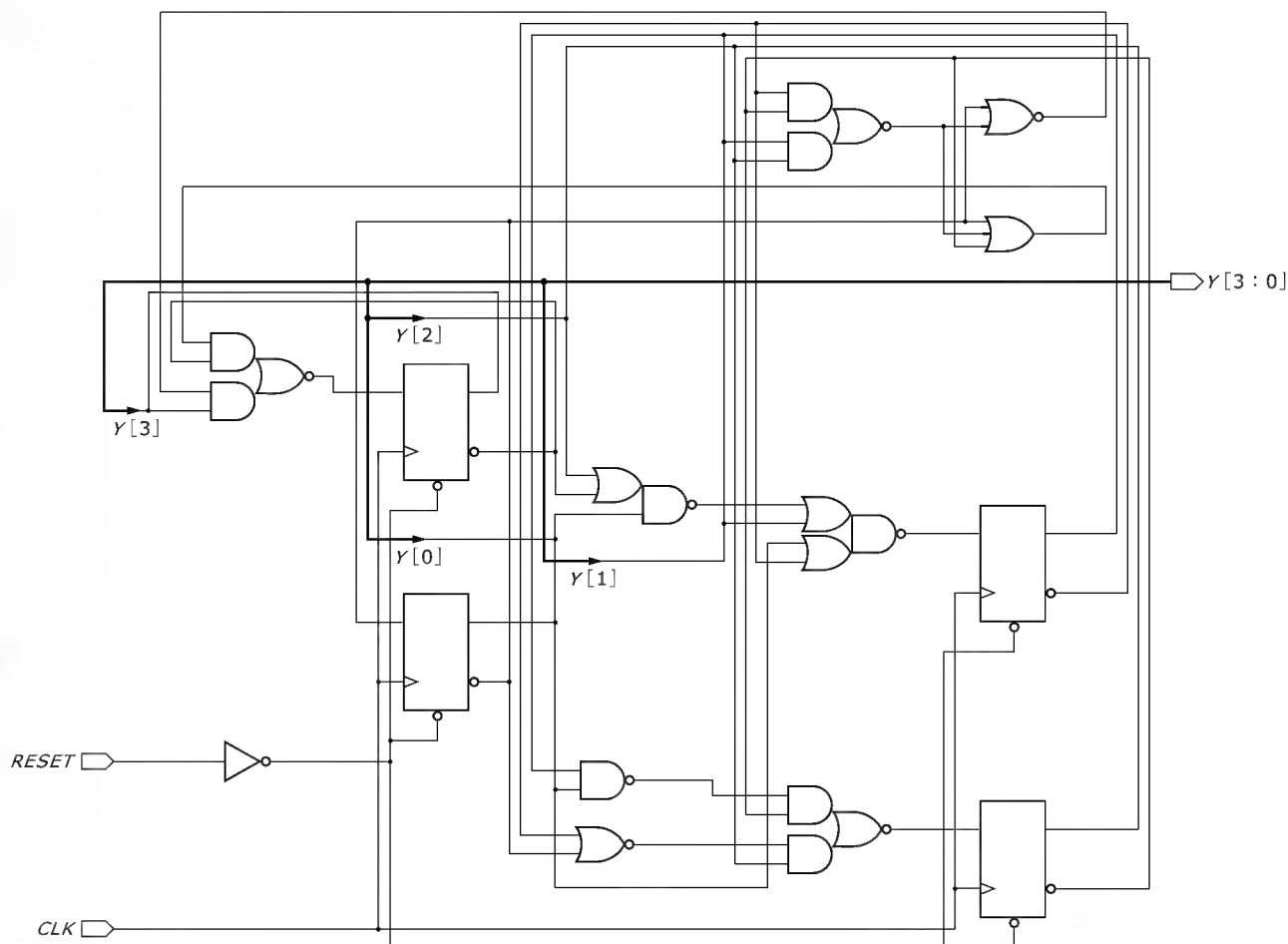
また，2^N 進アップダウンカウンタのシステムファイルを**リスト 15** に，16 進アップダウンカウンタのシミュレーション結果を**図 16** に示します．**図 16** から，制御信号 UD が 1 のときはカウントアップし，0 のときはカウントダウンしていることがわかります．

なお，**リスト 13** のヘッダファイルと**リスト 14** のインプリメンテーションファイルを Verilog-HDL 記述に変換し，それを論理合成した結果は，回路図が大きいため割愛します．

● リングカウンタの設計

これまでに紹介したカウンタは，カウント値を 2 進数で表現していました．このようなカウンタは，バイナリカウンタ（binary counter）と呼ばれます．しかし，実用上重要なカウンタには，バイナリカウンタ以外のカウンタ，すなわち非バイナリカウンタ（non-binary counter）も多数あります．ここでは，N 個の FF を縦続接続し，最終段の FF の出力を最初の段の FF

〔図 15〕 **10 進カウンタの合成結果**

Y[3:0]
Y[2]
Y[3]
Y[0]
Y[1]
RESET
CLK

へ入力することによって構成される，リングカウンタ(ring counter)と呼ばれるカウンタを設計します．

リングカウンタでは，N個の FF のうち，1 個の FF の値のみが 1 となって，残りの FF の値は 0 となります．たとえば，4 ビットリングカウンタ(4 個の FF を用いたリングカウンタ)の回路図およびタイミングチャートは**図 17** のようになります．**図 17(b)** に示すように，N個の FF を用いたリングカウンタの周期はNとなるため，NビットリングカウンタはN進カウンタとして動作することになります．

リングカウンタには，デコーダなどの回路が不要なため，回路構成が単純で，高速動作が可能であるという利点があります．一方，N個の FF を用いた場合，最大2^N進のカウンタを構成で

〔リスト 13〕2^N進アップダウンカウンタのヘッダファイルの記述例(up_down_counter.h)

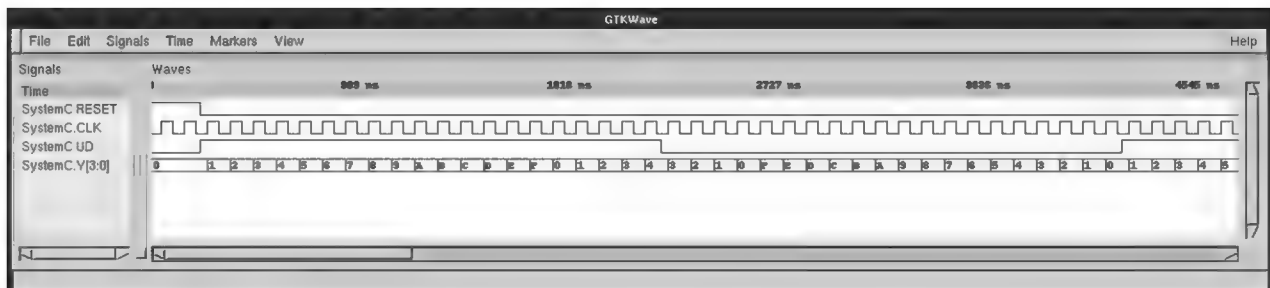
```
#include "systemc.h"

#define N  4

SC_MODULE(up_down_counter){
  sc_in_clk           CLK;
  sc_in<bool>         RESET;
  sc_in<bool>         UD;
  sc_out<sc_uint<N> > Y;

  void up_down_counter_behavior(void);

  SC_CTOR(up_down_counter){
    SC_METHOD(up_down_counter_behavior);
    sensitive << RESET.pos() << CLK.pos();
  }
};
```

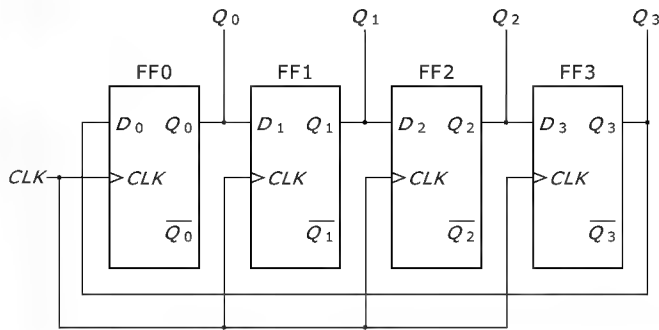
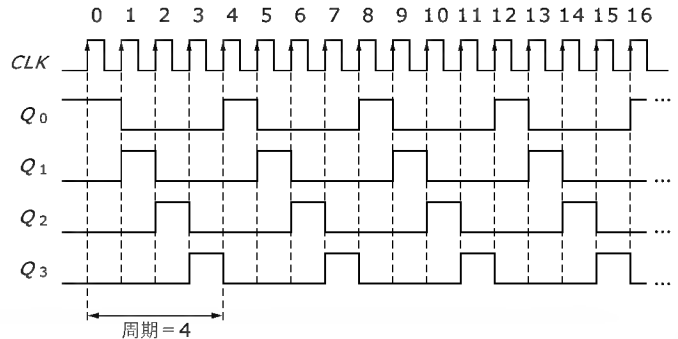
〔リスト 14〕2^N進アップダウンカウンタのインプリメンテーションファイルの記述例(up_down_counter.cpp)

```
#include "up_down_counter.h"

void up_down_counter::up_down_counter_behavior(void)
{
  if ( RESET.read() ) {
    Y = 0x0;
  } else {
    if ( UD == true ) {
      Y = Y.read() + 0x1;
    } else {
      Y = Y.read() - 0x1;
    }
  }
};
```

〔リスト 15〕2^N進アップダウンカウンタのシステムファイルの記述例(main_up_down_counter.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "up_down_counter.h"

int sc_main(int argc, char *argv[])
{
  int i;
  sc_clock CLK("CLK",100,SC_NS,0.5,0,SC_NS,false);
  sc_signal<bool>        RESET;
  sc_signal<bool>        UD;
  sc_signal<sc_uint<N> > Y;

  up_down_counter UP_DOWN_COUNTER("up_down_counter");

  UP_DOWN_COUNTER.CLK(CLK);
  UP_DOWN_COUNTER.RESET(RESET);
  UP_DOWN_COUNTER.UD(UD);
  UP_DOWN_COUNTER.Y(Y);

  sc_trace_file *trace_f;

  trace_f = sc_create_vcd_trace_file("up_down_counter_trace");
  ((vcd_trace_file *)trace_f)-> sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,CLK,"CLK");
  sc_trace(trace_f,RESET,"RESET");
  sc_trace(trace_f,UD,"UD");
  sc_trace(trace_f,Y,"Y");

  RESET = true;
  UD    = false;
  sc_start(220,SC_NS);
  RESET = false;
  for (i=0; i<10; i++) {
    if ( UD == true ) {
      UD = false;
    } else {
      UD = true;
    }
    sc_start(2000,SC_NS);
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

〔図 16〕16 進アップダウンカウンタのシミュレーション波形

制御信号 UD が 1 の場合はカウントアップし，0～2^Nを繰り返し出力している．また，制御信号 UD が 0 の場合はカウントダウンし，2^N～0 を繰り返し出力している．このことから，2^N進アップダウンカウンタとして動作していることがわかる．

〔図 17〕4 ビットリングカウンタ

(a) 回路図

(b) タイミングチャート

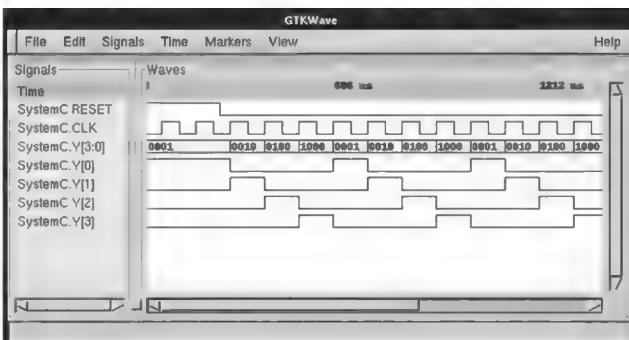
図(a)に示すように，Nビットリングカウンタは，N個のFFを縦続接続し，最終段のFFの出力を最初の段のFFへ入力することによって構成される．また図(b)に示すように，Nビットリングカウンタは，その周期がNとなるため，N進カウンタとして動作する．

〔リスト 16〕Nビットリングカウンタのヘッダファイルの記述例(ring_counter.h)

```
#include "systemc.h"

#define N  4

SC_MODULE(ring_counter){
  sc_in_clk           CLK;
  sc_in<bool>         RESET;
  sc_out<sc_uint<N> > Y;

  void ring_counter_behavior(void);

  SC_CTOR(ring_counter){
    SC_METHOD(ring_counter_behavior);
    sensitive << RESET.pos() << CLK.pos();
  }
};
```

〔リスト 17〕Nビットリングカウンタのインプリメンテーションファイルの記述例(ring_counter.cpp)

```
#include "ring_counter.h"

void ring_counter::ring_counter_behavior(void)
{
  if ( RESET.read() ) {
    Y = 0x1;
  } else {
    Y = (Y.read().range(N-2,0), Y.read()[N-1]);
  }
};
```

〔図 18〕4 ビットリングカウンタのシミュレーション波形

きますが，リングカウンタではN進カウンタしか構成できないため，FFの利用効率が良くないという欠点もあります．

まず，Nビットリングカウンタのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト 16**，**リスト 17**に示します．**リスト 16**のヘッダファイルでは，#defineを用いて，FFの数Nを変更できるようにしています．**リスト 16**ではNを4としているので，4ビットリングカウンタの記述になっています．また，**リスト 17**のインプリメンテーションファイルでは，リセット時に，出力Yの初期値として0x1を代入しています．リセット解除後は，この1が，次々に隣のFFにシフトされていきます．

なお，リングカウンタのシステムファイルは，インスタンス化の記述の部分を除いて，**リスト 12**に示したN進カウンタのシステムファイルと同じになるので，省略します．シミュレーション結果のみを**図 18**に示しておきます．**図 18**は，**図 17(b)**のタイミングチャートと同じに波形になっていることがわかります．

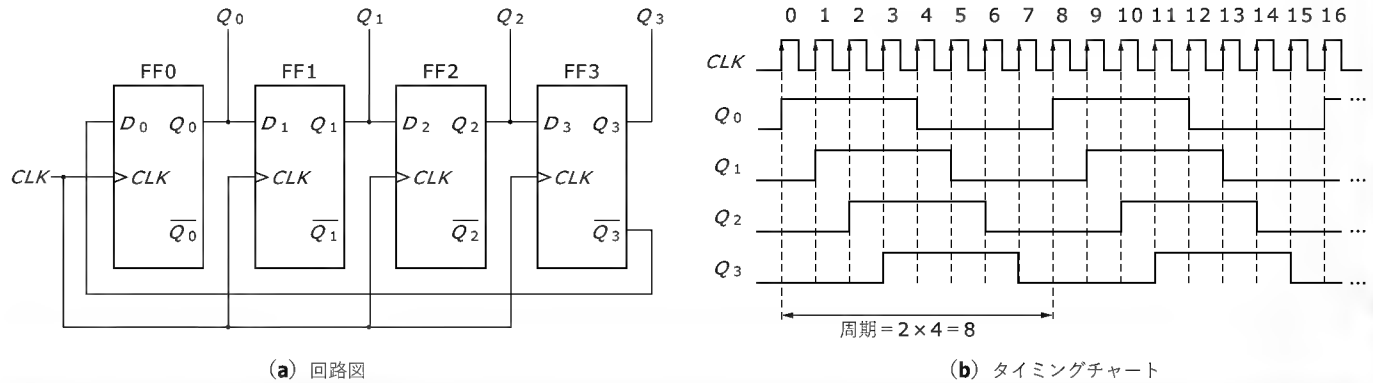
また，**リスト 16**のヘッダファイルと**リスト 17**のインプリメンテーションファイルをVerilog-HDL記述に変換し，それを論理合成した結果は，リセット信号の有無を除いて，**図 17(a)**の回路図と同じになるので，これも省略します．

● ジョンソンカウンタの設計

リングカウンタにおいて，最終段のFFの出力ではなく，最終段のFFの反転出力を最初の段のFFへ入力することによって構成される順序回路をジョンソンカウンタ(Johnson counter)と呼びます．このジョンソンカウンタも，よく用いられる非バイナリカウンタの一つです．

ジョンソンカウンタでは，1回のクロック入力で，ただ一つのFFの値しか変化しません．たとえば，4ビットジョンソンカウンタ(4個のFFを用いたジョンソンカウンタ)の回路図とタイミングチャートは**図 19**のようになります．**図 19(b)**に示すように，N個のFFを用いたジョンソンカウンタの周期は2N

〔図19〕4ビットジョンソンカウンタ

(a) 回路図　(b) タイミングチャート

図(a)に示すように，Nビットジョンソンソンカウンタは，N個のFFを縦続接続し，最終段のFFの反転出力を最初の段のFFへ入力することによって構成される．また図(b)に示すように，Nビットジョンソンソンカウンタは，その周期が2Nとなるため，2N進カウンタとして動作する．

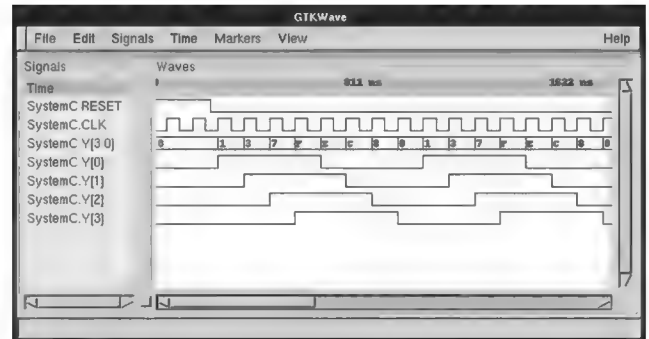
〔リスト18〕Nビットジョンソンソンカウンタのヘッダファイルの記述例(johnson_counter.h)

```
#include "systemc.h"

#define N  4

SC_MODULE(johnson_counter){
  sc_in_clk           CLK;
  sc_in<bool>         RESET;
  sc_out<sc_uint<N> > Y;

  void johnson_counter_behavior(void);

  SC_CTOR(johnson_counter){
    SC_METHOD(johnson_counter_behavior);
    sensitive << RESET.pos() << CLK.pos();
  }
};
```

となるため，Nビットジョンソンソンカウンタは2N進カウンタとして動作することになります．

ジョンソンカウンタは，リングカウンタと同様に，回路構造が単純で，高速動作が可能です．また，同時に複数の出力が変化することがないため，ハザードが生じにくい[注2]という特徴もあり，実際のディジタル回路設計においてよく用いられています．一方，ジョンソンカウンタは，リングカウンタと比べた場合，半分のFFで構成できますが，偶数進数のカウンタしか構成できないという欠点もあります．

まず，Nビットジョンソンソンカウンタのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト18**，**リスト19**に示します．**リスト18**のヘッダファイルでは，#defineを用いて，FFの数Nを変更できるようにしています．**リスト18**では，Nを4としているので，4ビットジョンソンソンカウンタの記述になっています．

なお，ジョンソンカウンタのシステムファイルは，インスタンス化の記述の部分を除いて，**リスト12**に示したN進カウンタのシステムファイルと同じになるので，省略します．シミュレーション結果のみを**図20**に示しておきます．**図20**は，**図19(b)**のタイミングチャートと同じに波形になっていることがわかります．

また，**リスト18**のヘッダファイルと**リスト19**のインプリメンテーションファイルをVerilog-HDL記述に変換し，それを論理合成した結果は，リセット信号の有無を除いて，**図19(a)**の回路図と同じになるので，これも省略します．

〔リスト19〕Nビットジョンソンソンカウンタのインプリメンテーションファイルの記述例(johnson_counter.cpp)

```
#include "johnson_counter.h"

void johnson_counter::johnson_counter_behavior(void)
{
  if ( RESET.read() == true ) {
    Y = 0x0;
  } else {
    Y = (Y.read().range(N-2,0), ~Y.read()[N-1]);
  }
};
```

〔図20〕4ビットジョンソンソンカウンタのシミュレーション波形

注2：詳細は割愛するが，ハザードは，回路内の複数の信号線の値が同時に変化する場合に生じやすくなる．

● グレイコードカウンタの設計

このほか，グレイコードカウンタ(Gray code counter)と呼ばれる非バイナリカウンタもよく用いられます．グレイコードカウンタは，その出力がグレイコード(Gray code：グレイ符号)になっているようなカウンタなので，まずグレイコードについて説明します．

グレイコードは2進数と1対1に対応する符号で，二つの2進数の値が1だけ異なる場合，それらの2進数に対応するグレイコードは1箇所だけ0と1が異なるという特徴をもっています．具体例として，4ビットの2進数に対するグレイコードの対応表を**表3**に示しておきます．また，4ビットの2進数$B = (B_3 B_2 B_1 B_0)$とグレイコード$G = (G_3 G_2 G_1 G_0)$を変換する回路を**図21**に示します．**図21**に示すように，2進数からグレイコードに変換するエンコーダおよびグレイコードから2進数に変換するデコーダは，どちらもXORゲートで構成することができます．

また，4ビットグレイコードカウンタ(4個のFFを用いたグレイコードカウンタ)のタイミングチャートを**図22**に示します．**図22**に示すように，N個のFFを用いたグレイコードカウンタの周期は2^Nとなるため，Nビットグレイコードカウンタは2^N進カウンタとして動作することになります．

グレイコードカウンタは，ジョンソンカウンタと同様に，同時に複数の出力が変化することがないため，ハザードが生じにくいという特徴があります．また，N個のFFを用いて2^N進カウンタを構成できるので，FFの利用効率も良く，実際のディジタル回路設計において良く用いられています．ただし，グレイコードカウンタは，リングカウンタやジョンソンカウンタと比べた場合，回路構造が複雑になってしまいます．

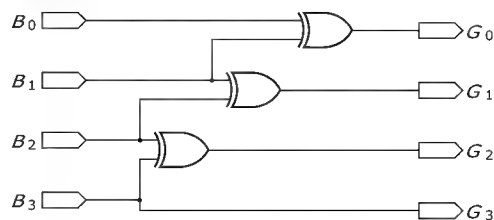
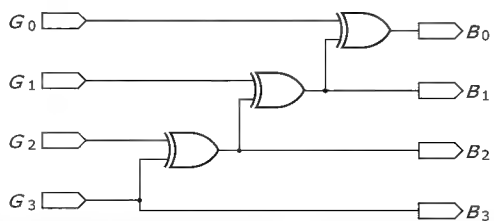
次にグレイコードカウンタを設計する方法について考えてみます．グレイコードカウンタには，グレイコードが保持されています．この保持されているグレイコードから，次のグレイコードを求めることができれば，グレイコードカウンタを設計できます．そこでまず，カウンタに保持されているグレイコードを，**図21(b)**の回路を用いて，一度，2進数に戻します．この2進数を1だけ増やしてから，**図21(a)**の回路を用いて，グレイコードに変換すれば，次のグレイコードが得られます．この方法を用いて設計すると，Nビットグレイコードカウンタのヘッダファイルとインプリメンテーションファイルは，それぞれ**リスト20**，**リスト21**のようになります．

リスト20のヘッダファイルでは，`#define`を用いて，FFの数Nを変更できるようにしています．**リスト20**では，Nを4としているので，4ビットグレイコードカウンタの記述になっています．

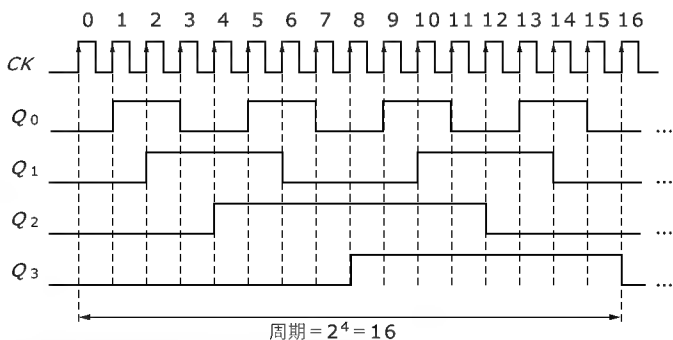
なお，システムファイルは，インスタンス化の記述の部分を除いて，**リスト12**に示したN進カウンタのシステムファイル

〔**表3**〕**4ビットの2進数とグレイコードの対応**

10進数	2進数	グレイコード
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

〔**図21**〕**2進数とグレイコードを変換する回路**

(**a**) 2進数-グレイコードエンコーダ

(**b**) グレイコード-2進数デコーダ

2進数からグレイコードに変換する2進数-グレイコードエンコーダ〔図(**a**)〕およびグレイコードから2進数に変換するグレイコード-2進数デコーダ〔**図(b)**〕は，どちらもXORゲートのみを用いて構成できる．

〔**図22**〕**4ビットグレイコードカウンタのタイミングチャート**

〔リスト20〕**Nビットグレイコードカウンタのヘッダファイルの記述例**(gray_code_counter.h)

```
#include "systemc.h"

#define F  4

SC_MODULE(gray_code_counter){
  sc_in_clk              CLK;
  sc_in<bool>            RESET;
  sc_out<sc_uint<F> >    Y;

  sc_signal<sc_uint<F> > COUNT;
  sc_signal<sc_uint<F> > GRAY;

  void count_up(void);
  void gray_code_gen(void);

  SC_CTOR(gray_code_counter){
    SC_METHOD(count_up);
    sensitive << RESET.pos() << CLK.pos();
    SC_METHOD(gray_code_gen);
    sensitive << Y;
  }
};
```

〔リスト21〕**Nビットグレイコードカウンタのインプリメンテーションファイルの記述例**(gray_code_counter.cpp)

```
#include "gray_code_counter.h"

void gray_code_counter::count_up(void)
{
  if ( RESET.read() ) {
    Y = 0x0;
  } else {
    Y = GRAY.read();
  }
};

void gray_code_counter::gray_code_gen(void)
{
  int i;
  sc_uint<F> TMP = Y.read();
  sc_uint<F> BIN;
  sc_uint<F> TMP_GRAY;

  BIN[F-1] = TMP[F-1];
  for (i=F-2;i>=0;i--) {
    BIN[i] = BIN[i+1] ^ TMP[i];
  }

  BIN++;

  TMP_GRAY[F-1] = BIN[F-1];
  for (i=F-2;i>=0;i--) {
    TMP_GRAY[i] = BIN[i+1] ^ BIN[i];
  }

  GRAY.write(TMP_GRAY);
};
```

現在のグレイコードから，次のグレイコードを得るために，まず，現在のグレイコードを**図21**(**b**)の回路を用いて，一度，2進数に戻している．次に，この2進数を1だけ増やしてから，**図21**(**a**)の回路を用いて，グレイコードに変換する．これによって，次のグレイコードが得られる．

と同じになるので省略し，シミュレーション結果のみを**図23**に示しておきます．**図23**は，**図22**のタイミングチャートと同じ波形になっていることがわかります．

また，**リスト20**のヘッダファイルと**リスト21**のインプリメンテーションファイルをVerilog-HDL記述に変換し，それを論理合成した結果も，回路図が大きいので割愛します．

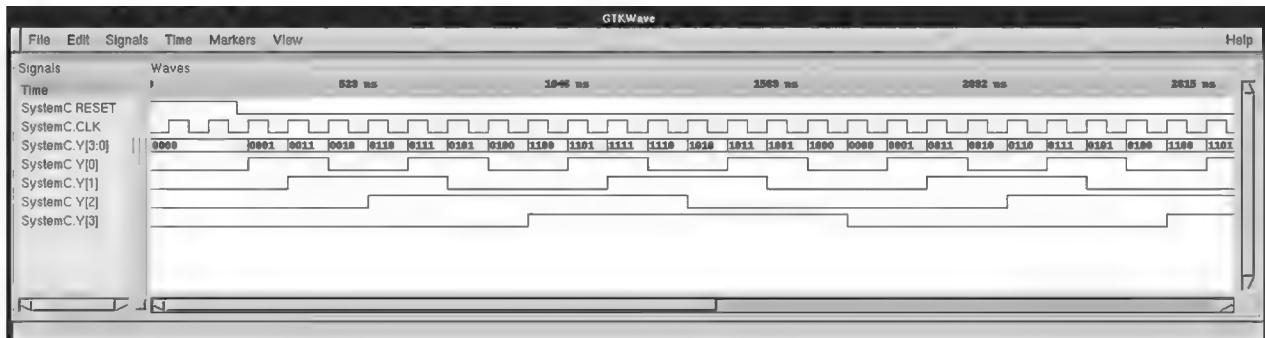
● ROMの設計

コンピュータ内部で用いられるメモリ(memory：記憶回路)には，先に紹介したレジスタのほかに，ROM(read only memory)やRAM(random access memory)などがあります．ROMは，読み込み専用のメモリです．ROMには，あらかじめ内部にデータが記憶されており，その内容を変更することはできません．一方，RAMは，読み込みだけでなく，書き込みも行えるメモリです．

ROMやRAMなどのメモリでは，8ビットや16ビットのデータを1ワード(word)とし，ワード単位でデータを保持します．8ビットのデータを1ワードとし，16ワードのデータを保持できるメモリの例を，**図24**に示します．**図24**のメモリでは，8ビット×16ワード＝128ビットのデータを保持できます．

図24に示すように，メモリに格納されている各データは，それらが格納されている場所，すなわちアドレス(address)によって区別されます．たとえば**図24**において，11(0BH)番地に格納されているデータは，00001111(0FH)です．なお，ここで，0BH，0FHは，それぞれ0B，0Fが16進数であることを表しています．

ここでは，**図25**に示すような入力線と出力線をもったROMを設計します．このROMは，クロック信号CLKの立ち上がりエッジで動作します．このとき，読み込み信号REが1であれ

〔図23〕**4ビットグレイコードカウンタのシミュレーション波形**

〔図24〕メモリにおけるデータとアドレス

アドレス(番地)	データ(8ビット)
0＝00H	01101100
1＝01H	10000000
2＝02H	01100010
3＝03H	00000000
4＝04H	10110001
5＝05H	10101010
6＝06H	00000001
7＝07H	00110011
8＝08H	00000100
9＝09H	01110100
10＝0AH	10000001
11＝0BH	00001111
12＝0CH	00000000
13＝0DH	00000000
14＝0EH	00000000
15＝0FH	00000000

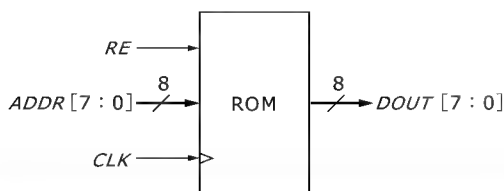
ROMやRAMなどのメモリでは，8ビットや16ビットのデータを1ワードとして，ワード単位でデータを保持する．各データが保持される場所は，アドレスによって区別される．図は，1ワードが8ビットで，16ワードのデータを保持できるメモリになっている．

〔リスト22〕**ROMのヘッダファイルの記述例**(rom.h)

```
#include "systemc.h"

#define ADDR_SIZE 8
#define WORD_SIZE 8

SC_MODULE(rom){
  sc_in_clk                    CLK;
  sc_in<bool>                  RE;
  sc_in<sc_uint<ADDR_SIZE> >   ADDR;
  sc_out<sc_uint<WORD_SIZE> >  DOUT;

  void rom_behavior(void);

  SC_CTOR(rom){
    SC_METHOD(rom_behavior);
    sensitive << CLK.pos();
  }
};
```

〔リスト23〕**ROMのインプリメンテーションファイルの記述例**(rom.cpp)

```
#include "rom.h"

void rom::rom_behavior(void)
{
  sc_uint<WORD_SIZE> TMP_D;

  if ( RE.read() == true ) {
    switch ( ADDR.read() ) {
      case 0x0 : TMP_D = 0x6C; break;
      case 0x1 : TMP_D = 0x80; break;
      case 0x2 : TMP_D = 0x62; break;
      case 0x3 : TMP_D = 0x43; break;
      case 0x4 : TMP_D = 0x76; break;
      case 0x5 : TMP_D = 0x29; break;
      case 0x6 : TMP_D = 0xF9; break;
      case 0x7 : TMP_D = 0x77; break;
      case 0x8 : TMP_D = 0xAD; break;
      case 0x9 : TMP_D = 0xA0; break;
      case 0xA : TMP_D = 0x0B; break;
      case 0xB : TMP_D = 0xEF; break;
      case 0xC : TMP_D = 0x00; break;
      case 0xD : TMP_D = 0x12; break;
      case 0xE : TMP_D = 0x33; break;
      case 0xF : TMP_D = 0x1F; break;
      default  : TMP_D = 0x00; break;
    }
  } else {
    TMP_D = 0x0;
  }

  DOUT.write(TMP_D);
};
```

図25 ROMの入力と出力

このROMは，クロック信号CLKの立ち上がりエッジで動作し，読み出し信号REが1であれば，アドレス信号ADDR[7:0]で指定されたアドレスに保持されている8ビットのデータが，データ出力DOUT[7:0]から出力される．一方，読み出し信号REが0の場合は，データ出力DOUT[7:0]から0が出力される．

ば，アドレス信号ADDR[7:0]で指定されたアドレスに保持されている8ビットのデータが，データ出力DOUT[7:0]から出力されます．読み込み信号REが0の場合は，データ出力DOUT[7:0]から0が出力されます．

図25に示したROMのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト22**，**リスト23**に示します．**リスト22**では，#defineを用いて，アドレス信号のビットサイズADDR_SIZEと，1ワードのビットサイズWORD_SIZEを変更できるようにしています．**リスト22**では，両方とも8にしているので，1ワードが8ビットで，256($=2^8$)ワードを保持できるROMになっています．また，**リスト23**では，case文を用いて，各アドレスのデータを指定しています．ROMのデータを増やしたり，記憶内容を変更する場合は，このcase文の内容を修正してください．

また，**図25**に示したROMのシステムファイルを，**リスト24**に，シミュレーション結果を**図26**に示します．なお，**リスト22**のヘッダファイルと**リスト23**のインプリメンテーションファイルをVerilog-HDL記述に変換し，それを論理合成した結果は，回路図が大きいため省略します．

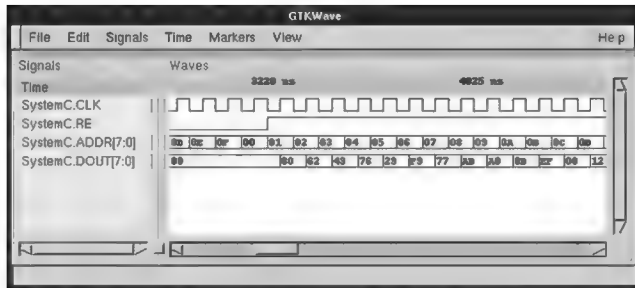
● RAMの設計

ここでは，**図27**に示すような入力線と出力線をもったRAMを設計します．このRAMは，クロック信号CLKの立ち上がりエッジで動作します．読み込み・書き込み信号RWが1の場合に読み出し，0の場合に書き込みをします．読み込みの場合，アドレス信号ADDR[7:0]で指定されたアドレスに保持されている8ビットのデータが，データ出力DOUT[7:0]から出力されます．一方，書き込みの場合，アドレス信号ADDR[7:0]で指定されたアドレスに，データ入力DIN[7:0]のデータが書き込まれます．このとき，データ出力DOUT[7:0]から0が出力されます．

図27に示したRAMのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト25**，**リスト26**に示します．**リスト25**では#defineを用いて，アドレス信号のビットサイズADDR_SIZEと，1ワードのビットサイズWORD_SIZEを変更できるようにしています．**リスト25**では，両方とも8にしている

〔リスト 24〕ROM のシステムファイルの記述例(main_rom.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "rom.h"

int sc_main(int argc, char *argv[])
{
  int i, j;
  sc_clock CLK("CLK",100,SC_NS,0.5,0,SC_NS,false);
  sc_signal<bool>               RE;
  sc_signal<sc_uint<ADDR_SIZE> > ADDR;
  sc_signal<sc_uint<WORD_SIZE> > DOUT;

  sc_uint<ADDR_SIZE> TMP_ADDR;

  rom ROM("rom");

  ROM.CLK(CLK);
  ROM.RE(RE);
  ROM.ADDR(ADDR);
  ROM.DOUT(DOUT);

  sc_trace_file *trace_f;

  trace_f = sc_create_vcd_trace_file("rom_trace");

  ((vcd_trace_file *)trace_f)-> sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,CLK,"CLK");
  sc_trace(trace_f,RE,"RE");
  sc_trace(trace_f,ADDR,"ADDR");
  sc_trace(trace_f,DOUT,"DOUT");

  RE       = false;
  TMP_ADDR = 0x0;
  for (i=0; i<10; i++) {
    if ( RE == true ) {
      RE = false;
    } else {
      RE = true;
    }
    for (j=0; j<16; j++) {
      if ( TMP_ADDR == 0xF ) {
        TMP_ADDR = 0x0;
      } else {
        TMP_ADDR++;
      }
      ADDR = TMP_ADDR;
      sc_start(100,SC_NS);
    }
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

〔図 26〕ROM のシミュレーション波形

クロック信号 CLK の各立ち上がりエッジで，読み込み信号 RE が 1 の場合に，アドレス信号 ADDR[7:0] で指定されたアドレスに保持されている 8 ビットのデータが，データ出力 DOUT[7:0] から出力されている．また，読み込み信号 RE が 0 の場合は，データ出力 DOUT[7:0] から 0 が出力されている．このことから，本文で説明した ROM として動作していることが確認できる．

〔図 27〕RAM の入力と出力

〔リスト 25〕RAM のヘッダファイルの記述例(ram.h)

```
#include "systemc.h"

#define ADDR_SIZE 8
#define WORD_SIZE 8
#define MEM_SIZE  256

SC_MODULE(ram){
  sc_in_clk                     CLK;
  sc_in<bool>                   RW;
  sc_in<sc_uint<ADDR_SIZE> >  ADDR;
  sc_in<sc_uint<WORD_SIZE> >  DIN;
  sc_out<sc_uint<WORD_SIZE> > DOUT;

  sc_uint<WORD_SIZE> MEM [MEM_SIZE];

  void ram_read_op(void);
  void ram_write_op(void);

  SC_CTOR(ram){
    SC_METHOD(ram_read_op);
    sensitive << CLK.pos();
    SC_METHOD(ram_write_op);
    sensitive << CLK.pos();
  }
};
```

ので，1 ワードが 8 ビットで，256 (＝ 2^8) ワードを保持できる RAM になっています．なお，**リスト 25** の MEM_SIZE は，配列のサイズを指定しているので，総ワード数を指定してください．この場合 ADDR_SIZE が 8 なので，256 (＝ 2^8) を指定しています．また**リスト 25**，**リスト 26** の記述では，RAM の読み出し動作と書き込み動作を別々のプロセスとして記述しています．

次に，**図 27** に示した RAM のシステムファイルを，**リスト 27** に，シミュレーション結果を**図 28** に示します．なお，**リスト 25** のヘッダファイルと**リスト 26** のインプリメンテーションファイルを Verilog-HDL 記述に変換し，それを論理合成した結果は，回路図が大きいため省略します．

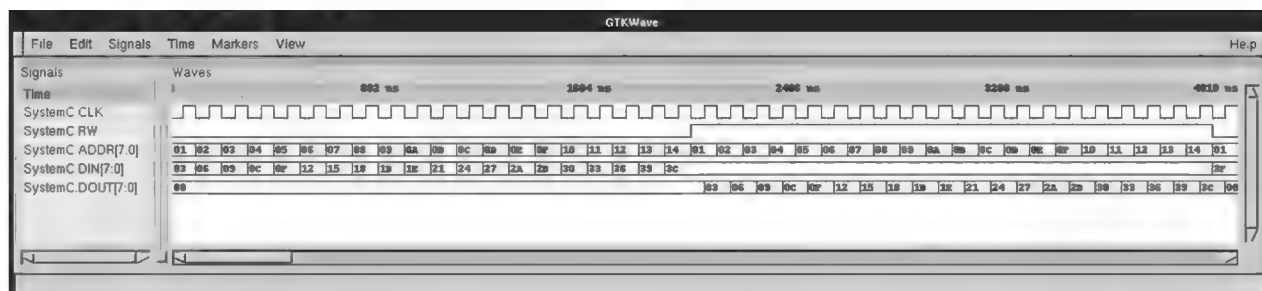
よしだ・たけお　琉球大学 工学部 情報工学

〔リスト26〕RAMのインプリメンテーションファイルの記述例(ram.cpp)

```
#include "ram.h"

void ram::ram_read_op(void)
{
  sc_uint<WORD_SIZE> TMP_DOUT;
  sc_uint<ADDR_SIZE> TMP_ADDR = ADDR.read();

  if ( RW.read() == true ) {
    TMP_DOUT = MEM[TMP_ADDR];
  } else {
    TMP_DOUT = 0x0;
  }
  DOUT.write(TMP_DOUT);
};

void ram::ram_write_op(void)
{
  sc_uint<WORD_SIZE> TMP_MEM = DIN.read();
  sc_uint<ADDR_SIZE> TMP_ADDR = ADDR.read();

  if ( RW.read() == false ) {
    MEM[TMP_ADDR] = TMP_MEM;
  }
};
```

〔リスト27〕RAMのシステムファイルの記述例(main_ram.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "ram.h"

int sc_main(int argc, char *argv[])
{
  int i, j;
  sc_clock CLK("CLK",100,SC_NS,0.5,0,SC_NS,false);
  sc_signal<bool>                 RW;
  sc_signal<sc_uint<ADDR_SIZE> > ADDR;
  sc_signal<sc_uint<WORD_SIZE> > DIN;
  sc_signal<sc_uint<WORD_SIZE> > DOUT;

  sc_uint<ADDR_SIZE> TMP_ADDR;
  sc_uint<WORD_SIZE> TMP_DIN;

  ram RAM("ram");

  RAM.CLK(CLK);
  RAM.RW(RW);
  RAM.ADDR(ADDR);
  RAM.DIN(DIN);
  RAM.DOUT(DOUT);

  sc_trace_file *trace_f;

  trace_f = sc_create_vcd_trace_file("ram_trace");

  ((vcd_trace_file *)trace_f)-> sc_set_vcd_time_unit(-9);
  sc_trace(trace_f,CLK,"CLK");
  sc_trace(trace_f,RW,"RW");
  sc_trace(trace_f,ADDR,"ADDR");
  sc_trace(trace_f,DIN,"DIN");
  sc_trace(trace_f,DOUT,"DOUT");

  TMP_DIN  = 0x0;
  for (i=0; i<10; i++) {
    TMP_ADDR = 0x0;
    for (j=0; j<20; j++) {
      RW = false;
      TMP_ADDR++;
      ADDR = TMP_ADDR;
      TMP_DIN = TMP_DIN + 0x03;
      DIN = TMP_DIN;
      sc_start(100,SC_NS);
    }

    TMP_ADDR = 0x0;
    for (j=0; j<20; j++) {
      RW = true;
      TMP_ADDR++;
      ADDR = TMP_ADDR;
      sc_start(100,SC_NS);
    }
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```

〔図28〕RAMのシミュレーション波形

クロック信号CLKの立ち上がりエッジで動作している．RAMの動作を確認するために，まず，読み込み・書き込み信号RWを0として，データの書き込みを行い，続いて，読み込み・書き込み信号RWを1として，書き込んだ内容を読み出している．書き込みの場合，アドレス信号ADDR[7:0]で指定されたアドレスに，データ入力DIN[7:0]のデータを書き込んでいる．このとき，データ出力DOUT[7:0]から0が出力されている．また，読み込みの場合，アドレス信号ADDR[7:0]で指定されたアドレスに保持されている8ビットのデータが，データ出力DOUT[7:0]から出力されている．以上のことから，本文で説明したRAMとして動作していることが確認できる．

Chapter 4

状態遷移する回路を表現する

ステートマシンのSystemC記述

吉田たけお

これまでの知識の総仕上げとして，ステートマシンをSystemCで記述する．ステートマシンは内部に状態をもち，その状態を遷移させることで複雑な事象を表現することが可能な，応用範囲の広い技術である．
本章では，このステートマシンのSystemC記述について解説を行う．

（編集部）

1　順序回路の表現と設計方法

● 順序回路のモデル

順序回路の設計に際しては，定式化された設計法が重要であると前述しましたが，順序回路の設計方法は，大きく2種類の方法に分けることができます．その一つは，第3章で紹介してきたレジスタ，カウンタなどの比較的簡単に実現できる順序回路および第2章で紹介した加算器，マルチプレクサなどの組み合わせ回路を用いて，より大きな順序回路を構築する方法です．この方法は定式化されていないものの，順序回路の大部分の設計に用いられています．もう一つの方法は，理論的な設計方法です．以降では，順序回路の理論的な設計方法について説明します．

何度か述べましたが，順序回路は記憶機能をもったディジタル回路，すなわち，FFを含んだディジタル回路です．このFF以外の部分は，組み合わせ回路になります．言い換えると，順序回路は，FFと組み合わせ回路により構成されたディジタル回路である，ということができます．このような順序回路は，**図1**に示すモデル（model）で表すことができます．これまでに紹介した順序回路を含めて，すべての順序回路は，理論的には**図1**に示すモデルのような構成で実現できます．

図1のモデルにおいて，$X=(X_0, X_1, \ldots, X_{l-1})$は入力信号，$Y=(Y_0, Y_1, \ldots, Y_{m-1})$は出力信号を表しています．また，$Q=(Q_0, Q_1, \ldots, Q_{n-1})$は内部状態（internal state）または単に状態（state）と呼ばれ，記憶回路（storage circuit）の出力を表しています．

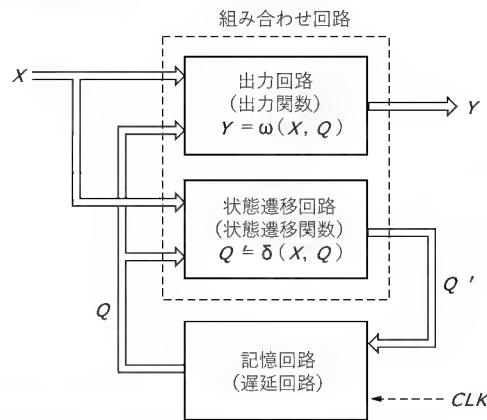
順序回路では，この「状態」と呼ばれる概念が重要になります．第2章でも述べましたが，組み合わせ回路では，現在の入力だけで出力が決定します．すなわち，どの時刻においても，同一の入力に対して同一の出力が得られます．一方，順序回路では，現在の入力と現在の状態によって出力が決定します．この状態の値は時刻によって変化します．このため順序回路では，異なる時刻においては，同一の入力に対して同一の出力が得られるとは限りません．

状態は記憶回路の出力であり，現時刻における入力と状態から，次の時刻の状態が決定されます．この，次の状態を決定するための組み合わせ回路を状態遷移回路（state transition circuit）と呼びます．また，状態遷移回路が実現している論理関数を状態遷移関数（state transition function）といい，δで表します．すなわち順序回路の次状態は，

$$Q' = \delta(X, Q) \quad \cdots\cdots(1)$$

によって決定されます．さらに，この状態遷移回路の出力は，記憶回路において次の時刻（次のクロック）まで保持されます．

また，順序回路の出力は現時刻の入力と状態によって決定されます．この出力を決定するための組み合わせ回路を出力回路（output circuit）といいます．また，出力回路が実現している論理関数を出力関数（output function）といい，ωで表します．

〔図1〕順序回路のモデル

順序回路は，記憶回路部と組み合わせ回路部から構成される．記憶回路部はFFなどの記憶機能をもった回路で構成される．また，組み合わせ回路部は，出力を決定するための出力回路および次状態を決定するための状態遷移回路から構成される．

すなわち順序回路の出力は,

$$Y = \omega(X,\ Q) \quad \cdots\cdots(2)$$

と表されます.

● 順序回路の表現

以上のように順序回路の動作は，状態遷移関数(状態遷移回路)と出力関数(出力回路)によって特徴付けられることになります．このような論理関数による表現の他に，直観的に理解しやすい図や表による表現も用いられます.

いま，順序回路の状態が$Q = q_i$であるとし，$X = x_k$が入力されたとき，$Y = y_h$が出力され，状態がq_jに変化するようすを**図2(a)**のように丸(○)と矢印で(→)表すことにします．また，順序回路の状態がq_iであるとし，x_lが入力されたときy_mが出力され，状態がq_iのままであるようすを**図2(b)**のように丸(○)とループで表すことにします．このとき，**図2**の記号を用いて，順序回路の状態変化および入出力のようすを表現した図を状態遷移図(state transition diagram)と呼びます.

たとえば，4進カウンタの状態遷移図は，**図3**のようになります．なお**図3**は，出力を00，01，10，11とした4進カウンタにはなっていません．**図3**はクロックのエッジで，入力Xが1である回数が4回になるたびに，出力Yから1を出力し，それ以外では0を出力するような4進カウンタの状態遷移図になっています．このように，順序回路の動作を，状態遷移図で表すことができます.

また順序回路の動作は，真理値表や特性表のような表を用いて表すこともできます．たとえば，**図3**の状態遷移図で表された4進カウンタの動作は，**表1**に示すような表を用いて表すこともできます．この表は，状態遷移表(state transition table)と呼ばれ，現状態における入力と，次状態および出力との対応表になっています．なお，状態遷移図と状態遷移表は，等価な表現なので，どちらを用いてもかまいません.

ところで，**図3**および**表1**では，各状態をq_0，q_1，...のように記号で表しています．状態は，**図1**のモデルにおける記憶回路の出力なので，本来は2進数で表すべきです．たとえば，**図3**および**表1**に示した4進カウンタの場合，q_0～q_3の四つの状態があるので，通常，2個のFFの出力Q_0，Q_1の組み合わせ{00，01，10，11}の4状態のいずれかが割り当てられます．このとき，FFの出力Q_0，Q_1を状態変数(state variable)と呼び，これらの状態変数に0または1の具体的な値を割り当てることを状態割り当て(state assignment)と呼びます.

例として，$q_0 = (00)$，$q_1 = (01)$，$q_2 = (10)$，$q_3 = (11)$と割り当てた場合の4進カウンタの状態遷移図および状態遷移表を，それぞれ**図4**および**表2**に示します.

● 順序回路の設計方法

以上のように順序回路の動作は，状態遷移図や状態遷移表で表せます．次に，これらの表現から，回路図を求める方法について説明します.

まず，**表2**に示した状態割り当て後の状態遷移表に着目します．**表2**においてX，Yは，それぞれ順序回路の入力と出力で

〔図2〕状態遷移図の書き方

(**a**) 状態が遷移する場合　(**b**) 状態が遷移しない場合

状態遷移図では，各状態を丸(○)で表す．また，状態が遷移するようすを矢印(→)で表す．さらに，矢印には，その遷移を起こす入力xと，そのときの出力yを，x/yとして付記する.

〔図3〕4進カウンタの状態遷移図

〔図4〕状態割り当て後の4進カウンタの状態遷移図

〔表1〕4進カウンタの状態遷移表

現状態 q	次状態 q' 入力X 0	1	出力Y 入力X 0	1
q_0	q_0	q_1	0	0
q_1	q_1	q_2	0	0
q_2	q_2	q_3	0	0
q_3	q_3	q_0	0	1

状態遷移表は，現状態における入力と，次状態および出力との対応表である.

〔表2〕状態割り当て後の4進カウンタの状態遷移表

現状態 Q_1	Q_0	次状態$Q'_1Q'_0$ 入力X 0		1		出力Y 入力X 0	1
0	0	0	0	0	1	0	0
0	1	0	1	1	0	0	0
1	0	1	0	1	1	0	0
1	1	1	1	0	0	0	1

〔表3〕4進カウンタの組み合わせ回路部の真理値表

X	Q_1	Q_0	Q'_0
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(a) 回路C1(Q'_0)の真理値表

X	Q_1	Q_0	Q'_1
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(b) 回路C2(Q'_1)の真理値表

X	Q_1	Q_0	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(c) 回路C3(Y)の真理値表

各表は，**表2**のQ'_0，Q'_1，Yを，それぞれX，Q_0，Q_1の関数として表した真理値表である．具体的には，**表2**の入力Xを，表の左側に移動しただけである．

す．また，**表2**では，状態変数を二つ用いているので，二つのFFを使用することになります．それらをFF0，FF1とします．このとき，**表2**のQ_0，Q_1は，それぞれFF0，FF1が，現在の時刻で保持している値になります．また，Q'_0，Q'_1は，それぞれFF0，FF1が次の時刻で保持する値になります．

このとき，次状態Q'_0，Q'_1および出力Yは，それぞれ，現在の入力Xおよび現在の状態Q_0，Q_1の論理関数として表すことができます．このことは，**表2**を，**表3**に示す三つの真理値表に書き換えてみるとわかりやすくなります．このとき，**表3**(**a**)，(**b**)の真理値表を実現する論理関数が，**表2**の4進カウンタの状態遷移関数になります．また，**表3**(**c**)の真理値表を実現する論理関数が，出力関数になります．ここではとりあえず，**表3**(**a**)のQ'_0を求める組み合わせ回路をC1，**表3**(**b**)のQ'_1を求める組み合わせ回路をC2，**表3**(**c**)のYを求める組み合わせ回路をC3，と表すことにします．このとき，これらの組み合わせ回路と二つのD-FF(FF0，FF1)を用いて，**図5**に示すような回路を構成することによって，**表2**の4進カウンタを実現できます．

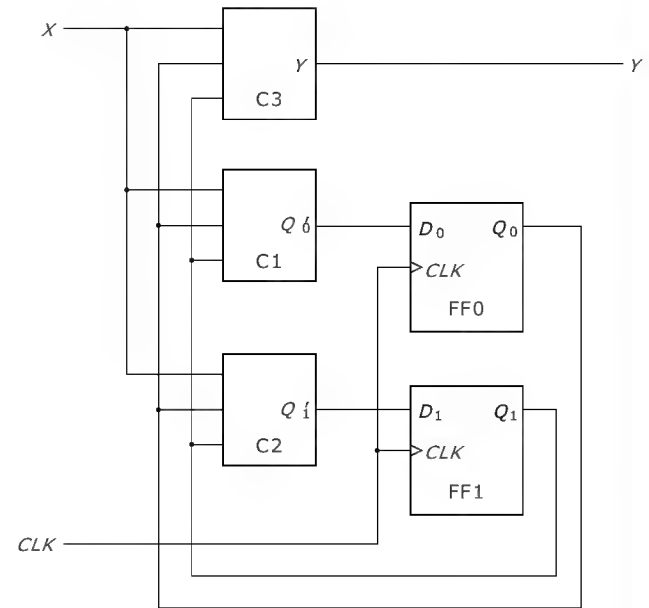
以上のことから，組み合わせ回路C1，C2，C3をゲート回路で構成できれば，4進カウンタの回路図が得られることになります．ここでは，導出過程の詳細は割愛しますが，**表3**の各表から，以下の論理関数が得られます．

$$Q'_0 = \delta_0(X, Q_0, Q_1) = \overline{X} \cdot Q_0 + X \cdot \overline{Q_0} \quad \cdots\cdots(3)$$

$$Q'_1 = \delta_1(X, Q_0, Q_1) = \overline{X} \cdot Q_1 + \overline{Q_0} \cdot Q_1 + X \cdot Q_0 \cdot \overline{Q_1} \quad \cdots\cdots(4)$$

$$Y = \omega(X, Q_0, Q_1) = X \cdot Q_0 \cdot Q_1 \quad \cdots\cdots(5)$$

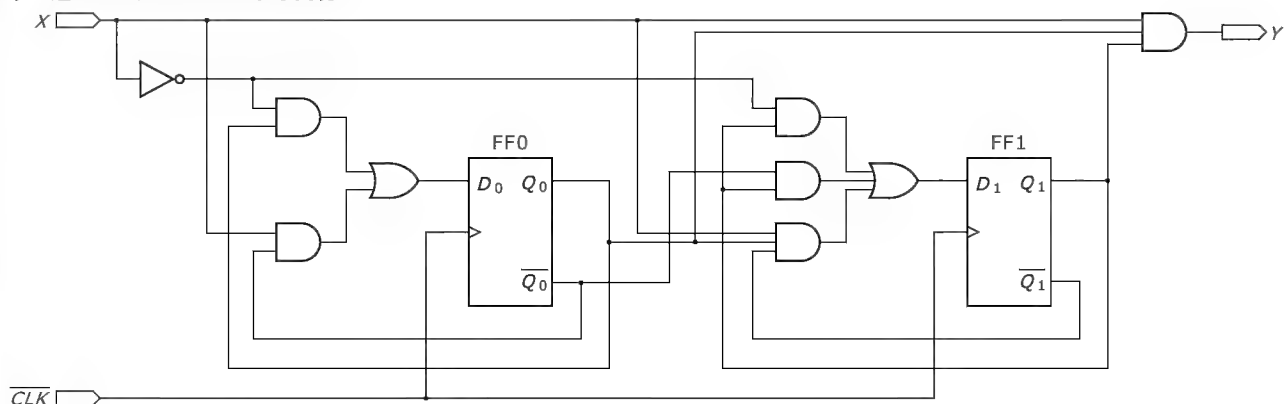
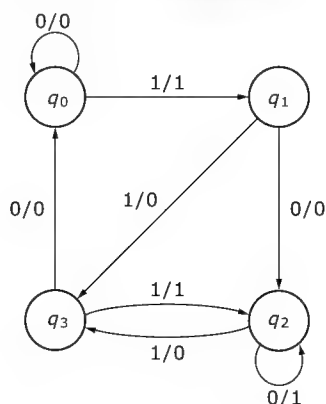
ここで，式(3)のδ_0および式(4)のδ_1が**表2**の4進カウンタの状態遷移関数になり，式(5)のωが出力関数になります．また，これらの状態遷移関数，出力関数を用いると，**表2**の4進カウンタは，**図6**のようになります．なお**図6**の回路では，D-FFの反転出力を利用して，構成を簡単にしています．

〔図5〕D-FFを用いた4進カウンタの構成

2 ステートマシンとその設計

● ステートマシンとは？

以上で説明した順序回路の設計方法は，状態遷移図あるいは状態遷移表から，状態遷移関数および出力関数を求め，これらの論理関数をもとに回路図を作成する，という手順になります．このように，状態遷移図(あるいは状態遷移表)から設計された順序回路を，とくにステートマシン(state machine)と呼んでいます．

順序回路は多くの場合，算術演算や論理演算などのデータ処理を中心に行うデータパス(data path)と，データパスを制御する制御回路(controller)に分けて設計されます．前者のデータパスは，それを構成する部品の多くが設計済みの回路を再利用できるため，ほとんどの場合，先に述べたもう一つの設計方法で設計されます．一方，後者の制御回路は，多くの場合，設計済みの回路を再利用できず，新たに設計する必要があ

〔図6〕4進カウンタのD-FFによる実現

〔図7〕設計するステートマシンの状態遷移図

〔図8〕ステートマシンのシミュレーション波形

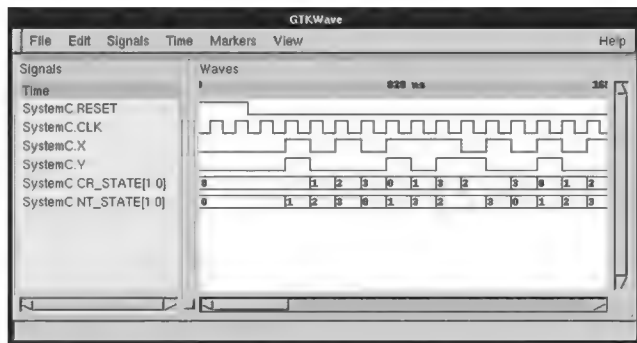



図7の状態遷移図のとおりに動作していることが確認できる．

るため，ステートマシンとして設計されます．

このようにステートマシンでは，制御回路としての役割が重要となっています．

● ステートマシンの設計

ここでは，ステートマシンをSystemCで記述する方法について見てみましょう．カウンタは，通常，第3章で示した**リスト10**，**リスト11**のように記述するので，先に示した4進カウンタではなく，**図7**に示す状態遷移図からステートマシンを設計することにします．

SystemCでステートマシンを記述する場合，**図1**のモデルにおける記憶回路部と組み合わせ回路部を別々のプロセスとして記述します．まず，**図7**のステートマシンのヘッダファイルとインプリメンテーションファイルを，それぞれ**リスト1**，**リスト2**に示します．

リスト1のヘッダファイルには，記憶回路部のプロセスsm_regと，組み合わせ回路部のプロセスsm_ccが記述されています．プロセスsm_regは，レジスタ(順序回路)なので，クロック信号CLKと，クロック信号より優先されるリセット信号RESETが，センシティビティリストに指定されています．また，プロセスsm_ccは，組み合わせ回路であり，その入力は，入力Xと現状態CR_STATEです．そのため，これらの信号がセンシティビティリストに指定されています．なお，現状態CR_STATEは内部信号線として，ヘッダファイル内で宣言しています．

一方，**リスト2**のインプリメンテーションファイルでは，プロセスsm_regとプロセスsm_ccの動作を記述しています．プロセスsm_regの部分は，レジスタの記述と同じであり，クロックの立ち上がりエッジのたびに，次状態NT_STATEを現状態CR_STATEに代入しています．この次状態NT_STATEも内部信号線として，ヘッダファイル内で宣言しています．また，プロセスsm_ccの部分では，case文を用いて，各状態ごとに，次状態と出力を定義しています．

リスト1，**リスト2**を見ればわかると思いますが，ステートマシンの状態遷移図が与えられれば，そのSystemC記述を機械的に得ることができます．

次に，**図7**のステートマシンのシステムファイルを**リスト3**に，シミュレーション結果を**図8**に示します．なお，**リスト1**のヘッダファイルと**リスト2**のインプリメンテーションファイルをVerilog-HDL記述に変換し，それを論理合成した結果を**図9**に示します．

〔リスト1〕ステートマシンのヘッダファイルの記述例(state_machine.h)

```
#include "systemc.h"

#define Q0 0x0
#define Q1 0x1
#define Q2 0x2
#define Q3 0x3

SC_MODULE(state_machine){
  sc_in_clk    CLK;
  sc_in<bool>  RESET;
  sc_in<bool>  X;
  sc_out<bool> Y;

  sc_signal<sc_uint<2> > CR_STATE;
  sc_signal<sc_uint<2> > NT_STATE;

  void sm_reg(void);
  void sm_cc(void);

  SC_CTOR(state_machine){
    SC_METHOD(sm_reg);
    sensitive << RESET.pos() << CLK.pos();
    SC_METHOD(sm_cc);
    sensitive << CR_STATE << X;
  }
};
```

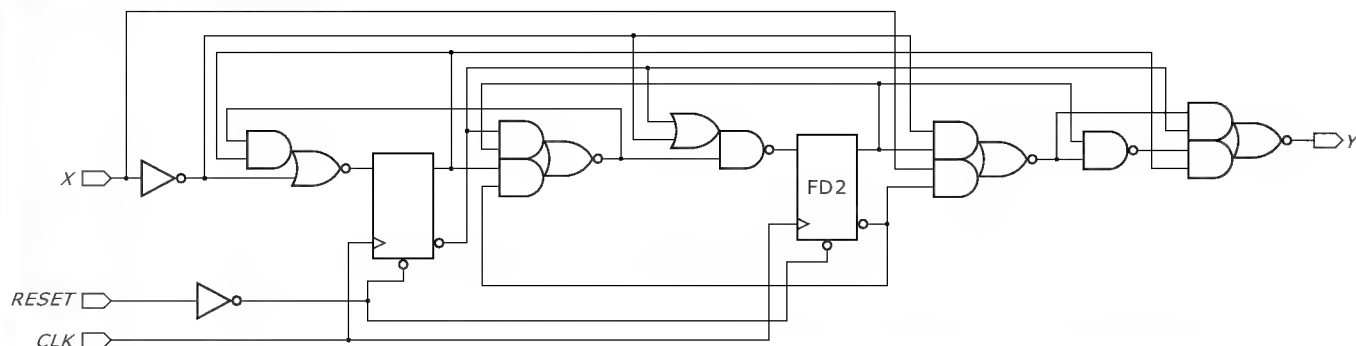
〔リスト2〕ステートマシンのインプリメンテーションファイルの記述例(state_machine.cpp)

```
#include "state_machine.h"

void state_machine::sm_reg(void)
{
  if ( RESET.read() ) {
    CR_STATE = Q0;
  } else {
    CR_STATE = NT_STATE.read();
  }
};

void state_machine::sm_cc(void)
{
  sc_uint<2> TMP_NS;
  bool       TMP_X;
  bool       TMP_Y;

  TMP_X = X.read();

  switch ( CR_STATE.read() ) {
    case Q0 : if ( TMP_X ) {
                TMP_NS = Q1;
                TMP_Y  = true;
              } else {
                TMP_NS = Q0;
                TMP_Y  = false;
              } break;
    case Q1 : if ( TMP_X ) {
                TMP_NS = Q3;
                TMP_Y  = false;
              } else {
                TMP_NS = Q2;
                TMP_Y  = false;
              } break;
    case Q2 : if ( TMP_X ) {
                TMP_NS = Q3;
                TMP_Y  = false;
              } else {
                TMP_NS = Q2;
                TMP_Y  = true;
              } break;
    default : if ( TMP_X ) {
                TMP_NS = Q2;
                TMP_Y  = true;
              } else {
                TMP_NS = Q0;
                TMP_Y  = false;
              } break;
  }

  Y.write(TMP_Y);
  NT_STATE.write(TMP_NS);
};
```

〔リスト3〕ステートマシンのシステムファイルの記述例(main_state_machine.cpp)

```
#include <stdio.h>
#include <iostream>
#include "systemc.h"
#include "state_machine.h"

int sc_main(int argc, char *argv[])
{
  int i;
  sc_clock CLK("CLK",100,SC_NS,0.5,0,SC_NS,false);
  sc_signal<bool> RESET;
  sc_signal<bool> X;
  sc_signal<bool> Y;

  state_machine STATE_MACHINE("state_machine");

  STATE_MACHINE.CLK(CLK);
  STATE_MACHINE.RESET(RESET);
  STATE_MACHINE.X(X);
  STATE_MACHINE.Y(Y);

  sc_trace_file *trace_f;

  trace_f = sc_create_vcd_trace_file("state_machine_trace");

  ((vcd_trace_file *)trace_f)-> sc_set_vcd_time_unit(-9);

  sc_trace(trace_f,CLK,"CLK");
  sc_trace(trace_f,RESET,"RESET");
  sc_trace(trace_f,X,"X");
  sc_trace(trace_f,Y,"Y");
  sc_trace(trace_f,STATE_MACHINE.CR_STATE,"CR_STATE");
  sc_trace(trace_f,STATE_MACHINE.NT_STATE,"NT_STATE");

  RESET = true;
  X     = false;
  sc_start(200,SC_NS);
  RESET = false;
  sc_start(50,SC_NS);
  for (i=0; i<10; i++) {
    X = false;
    sc_start(100,SC_NS);
    X = true;
    sc_start(100,SC_NS);
    X = false;
    sc_start(100,SC_NS);
    X = true;
    sc_start(100,SC_NS);
    X = false;
    sc_start(100,SC_NS);
    X = true;
    sc_start(300,SC_NS);
  }

  sc_close_vcd_trace_file(trace_f);

  return 0;
}
```


News Flash ── Key3Media Group, 社名変更で Medialive International に
Key3Media Group(キースリーメディア・グループ)は, 2月に申請した米国破産法における財務健全化計画を完了し, あらたに社名を Medialive International(メディアライブ・インターナショナル)とする. キースリーメディア・グループは, COMDEX の主催会社として知られる.

〔図9〕ステートマシンの合成結果

SystemCのヘッダファイル(**リスト1**)とインプリメンテーションファイル(**リスト2**)をVerilog-HDLへ変換してから，それを論理合成すると，このような回路図が得られる．

まとめ

第2章から第4章では，ディジタル回路の基礎およびディジタル回路をSystemCで記述する方法について解説してきました．これらの章では，可能な限りC++言語の用語を用いずに，できるだけ多くの記述例を紹介するよう心がけました．また，紹介したディジタル回路の記述は，論理合成可能なRTLモデルのSystemC記述になっており，実際に論理合成し，その結果を確認してきました．第2章から第4章までの内容で，SystemCのRTLモデルの記述方法について，その基礎を理解していただけたのではないかと思います．

なお，今回は見た目のわかりやすさを優先したので，第2章から第4章で示した記述例の中には，より簡潔に記述できるものがたくさんあります．これからSystemCを本格的に学ぼうと思っている方は，より簡潔な記述にも挑戦してみてください．SystemCの記述方法はバリエーションが豊富なので，今回紹介できなかった新しい記述方法を発見できると思います．

よしだ・たけお　琉球大学 工学部 情報工学

FPGAとDSPを
搭載したボードの設計事例

SpecCによる協調設計の実際

田中康一郎

これまでの章ではC/C++ベースシステム設計言語について，さまざまな切り口から解説してきた．
そこで本章では，これまでの知識を元に，C/C++ベースシステム設計言語の一つであるSpecCを使って，FPGAとDSPを搭載したボードの設計を行う．従来まったく別の言語で記述していたハードウェアとソフトウェアだが，SpecCを介することによりこれらの協調設計が可能になる例として参考になるであろう．
（編集部）

はじめに

LSI設計の方法は，回路図エディタを使った論理回路による設計から，HDLによるレジスタトランスファレベル（RTL）の設計に変わりつつあります．これにより，昔と比べて一つのLSIを設計することは非常に容易になりました．しかし多くの組み込みシステムでは組み込みプロセッサやDSPを利用しているため，システム全体を開発するには何種類かの異なる言語を利用する必要があります．

また，そのようなシステムでは，ハードウェアとソフトウェアのトレードオフがその性能に大きく影響しますが，いくつかの異なる言語で開発されたモデルはトレードオフに非常に手間がかかるため，システムの開発期間が長くなる傾向があります．開発期間が延びると製品の商品価値が大きく損なわれるため，システム設計者にとってはきわめて重要な問題です．

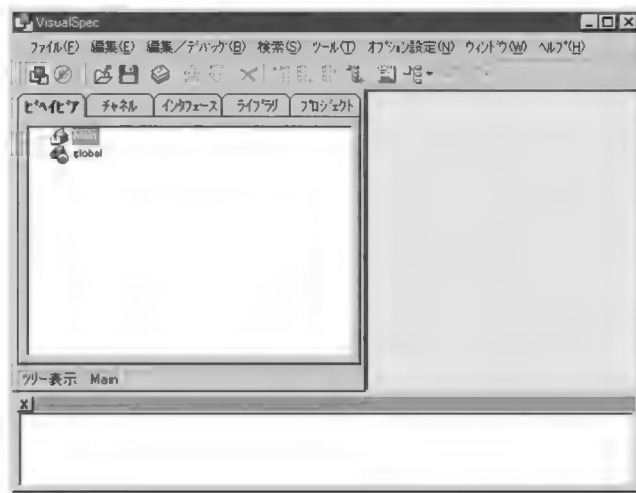
上記のような問題を解決するために，システムレベル設計言語（SLDL）と総称される言語を利用する研究がさかんに行われています．SLDLはHDLに変わる次世代のシステムLSI設計言語であり，HDLとは異なり，ハードウェアとソフトウェアを同時に区別することなく記述することができます．有名なSLDLには，**SystemC**と**SpecC**があります．名称が似ているため，類似した言語と思われがちですが，思想が大きく異なるため，その詳細はかなり違います．

SystemCは，プログラミング言語の一つであるC++を，SpecCはプログラミング言語の一つであるCをベースにしています．どちらも同一環境でハードウェアとソフトウェアを同時に開発することができます．しかしSystemCでは，ハードウェアをデザインする場合，明らかに意識して設計入力する必要があります．他方，SpecCはCに対してハードウェアを構成しやすいような拡張をすることで実現しており，SpecCはSystemCよりもハードウェアとソフトウェアのトレードオフを行うことが容易な環境を提供できるSLDLであるといえます．

この章では，前章までで説明してきたSystemCではなく，もう一つのSLDLであるSpecCによる協調設計検証の事例を紹介します．この事例では，市販されているSpecCをベースとしたEDAツールを利用して，ハードウェアとソフトウェアが混在したシステムで簡単な画像処理アプリケーションの開発を行います．ターゲットとなるハードウェア環境は，筆者らの大学で開発したものです．

SpecC言語を利用したEDAツール

現在，SpecC言語を利用したEDAツールは2種類存在します．一つは，米国のUniversity of California, Irvine校で開発され，SpecC Technology Open Consortium（STOC）からフリーソフトウェアとして公開されているSpecCのリファレンスコンパイラ（SCRC），もう一つは，InterDesign Technologies（IDT）社から販売されているVisualSpecです（**図1**）．

〔図1〕IDT VisualSpec

SCRCでは，SpecCで記述したコードをシミュレーションすることができます．しかし，残念ながらハードウェアの設計を行うことはできません．一方，VisualSpecは上流仕様設計ツールとして位置づけられており，SpecCをすべてテキストで記述するというスタイルではなく，動作（ビヘイビア）はテキストエディタで入力し，接続関係などのシステム構成に関することはビジュアル的に入力できる環境を提供しています．これにより，VisualSpecではSpecCの詳細を知らないCプログラマでもモデル設計が行えます．また，ハードウェアとソフトウェアの双方を同時にモデリング・検証し，その後それらの設計データからハードウェア用のコードとソフトウェア用のコードを生成することができます．

VisualSpecによるRTL生成までの流れ

VisualSpecを利用した設計フローを紹介しましょう（**図2**）．

モデルの設計入力終了後，VisualSpecではそのモデルのシミュレーションを行います．このシミュレーションでは，入力したモデルのデータからSpecCコードを生成し，そのコードをコンパイルすることで行われます．このコンパイラは，SCRCとは異なるSCCという別のコンパイラを使います．

シミュレーションが終了すると，ハードウェアとソフトウェアとの分割を行います．分割の単位は，SpecCでいうビヘイビア，つまり動作で行われ，あるビヘイビアはハードウェアにするかそれともソフトウェアにするかを選択していきます．その後，ソフトウェアとして選択されたビヘイビアは，それらだけの別プロジェクトとして生成されます．このプロジェクトから必要に応じてソフトウェア向けのコードを生成することができます．

一方，ハードウェア向けのSpecCコードはCコードに変更されます．VisualSpecでは，残念ながらレジスタトランスファレベルのハードウェア記述言語（RTL）を直接生成することができません．しかし，Y Explorations（日本代理店はソリトンシステムズ）から販売されているeXCiteという高位合成ツール向けのCコードが生成できるので，VisualSpecとeXCiteの両方を利用することでRTLを生成できます．

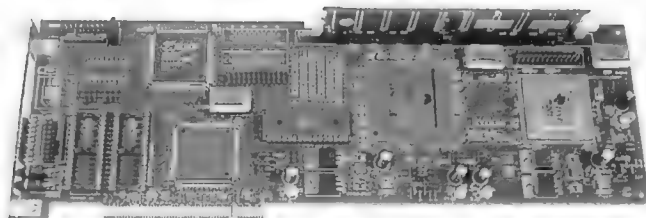
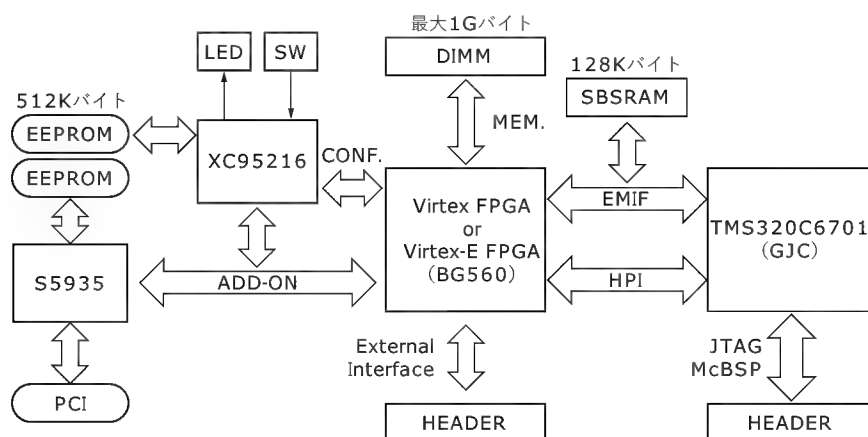
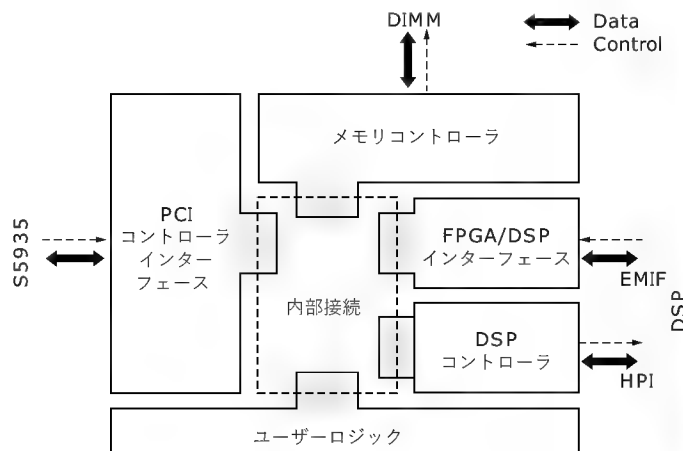
〔図2〕VisualSpecを利用したRTL生成までの流れ

「RYUOH」：FPGAとDSPを載せたPCIカード

筆者の勤める九州工業大学では，Field Programmable Gate Array（FPGA）をベースとした研究を行っています．その研究の一環として，FPGAとDSPを載せたPCIカードである「RYUOH」（**図3**）を開発しました．FPGAとDSPを組み合せたのは，今後多くのシステムがハードウェアとソフトウェア（プロセッサ）を混在させたシステム上に構築されるようになることが予想される中で，とくにFPGAのようなハードウェア的なプログラマビリティをもつデバイスとプロセッサのようなソフトウェア的なプログラマビリティをもつデバイスによるハードウェア/ソフトウェア協調設計が重要であるという理由からでした．なお，このRYUOHは，基板の製造以外はすべて大学の学生たちとともに設計実装した手作りのハードウェアです．

RYUOHには，いくつかの主要なデバイスが載せてあります．FPGAには，Xilinx社のVirtex（-E）シリーズの560ピンパッケージのものが利用できます（FPGAコアに対する電源電圧の切り替えも可能）．また，DSPには，TI社のTMS320C6xシリーズ（'C6x）が利用できます．さらに，メモリとしてPC向けのDIMM（PC133）が利用できます．これとは別に，'C6x用ローカルメモリとして128KバイトのSBSRAM（Synchronous Burst SRAM）が実装してあります．

RYUOHの主要デバイスは，**図4**のように接続されています．'C6xの主要な外部インターフェースには，メモリを接続するためのExternal Memory Interface（EMIF）と，'C6xを制御するためのHost Port Interface（HPI）の2種類がありますが，これらのどちらも接続しています．通常の場合，DSPと外部ロジックの通信はHPIを使って通信しますが，RYUOHはFPGA内にDSPのEMIFと通信できるロジックを実装することで通信できるようにしています．また，DIMMはSDRAMの集合体ですから，FPGA内にSDRAMコントローラ＋αの機能を実現

〔図3〕RYUOH：FPGA/DSP Based PCI Card

〔図4〕RYUOHの接続構成

することで通信が行えます．

RYUOHは，PCIカードですからPCから通信するためには，ドライバが必要になります．筆者らはRYUOH用ドライバとして，Windows XP用とWindows NT 4.0用のドライバも開発しました．

RYUOH用HDLライブラリ

RYUOHは前節で説明したとおり，すべてのデバイスがFPGAに接続されているので，各デバイスを利用するためにはそれぞれを制御する回路が必要となります．そこで，これらの制御回路をHDLライブラリとして提供しています．**図5**にFPGAに対する各機能を示します．このライブラリ群は，'C6xのHPIを経由してDSPを制御するためのDSPコントローラ，'C6xのEMIFを利用してFPGAとDSPが通信するためのFPGA/DSPインターフェース，DIMMを利用するためのメモリコントローラ，PCIバスを経由してPCと通信するためのPCIコントローラインターフェースから構成されています．また，これらは各デバイス制御機構だけでなく，ほかの制御機構やユーザーロジック（設計者がアプリケーションを実現する部分）とのインターフェース回路と調停機構も含んでいます．以下，各制御機構を簡単に紹介します．

● DSPコントローラ

'C6xの動作モード設定，内部状態の観測，プログラムブートなどを行うモジュールです．これらはHPIを経由して'C6xのメモリ空間にマッピングされた制御レジスタにアクセスすることによって実現しています．動作速度の設定などの一部機能は専用信号を用いますが，ほとんどの機能は単なるメモリアクセスとして行われます．ライブラリでは，各設定項目は`define`文で定義しているため，各ユーザーがそれらの設定を簡単に変更することもできます．HPIはバス幅16ビットの非同期インターフェースなので，32ビットのデータを2回の16ビットデータ

〔図5〕FPGA内部の構成

に分割して転送を行います．

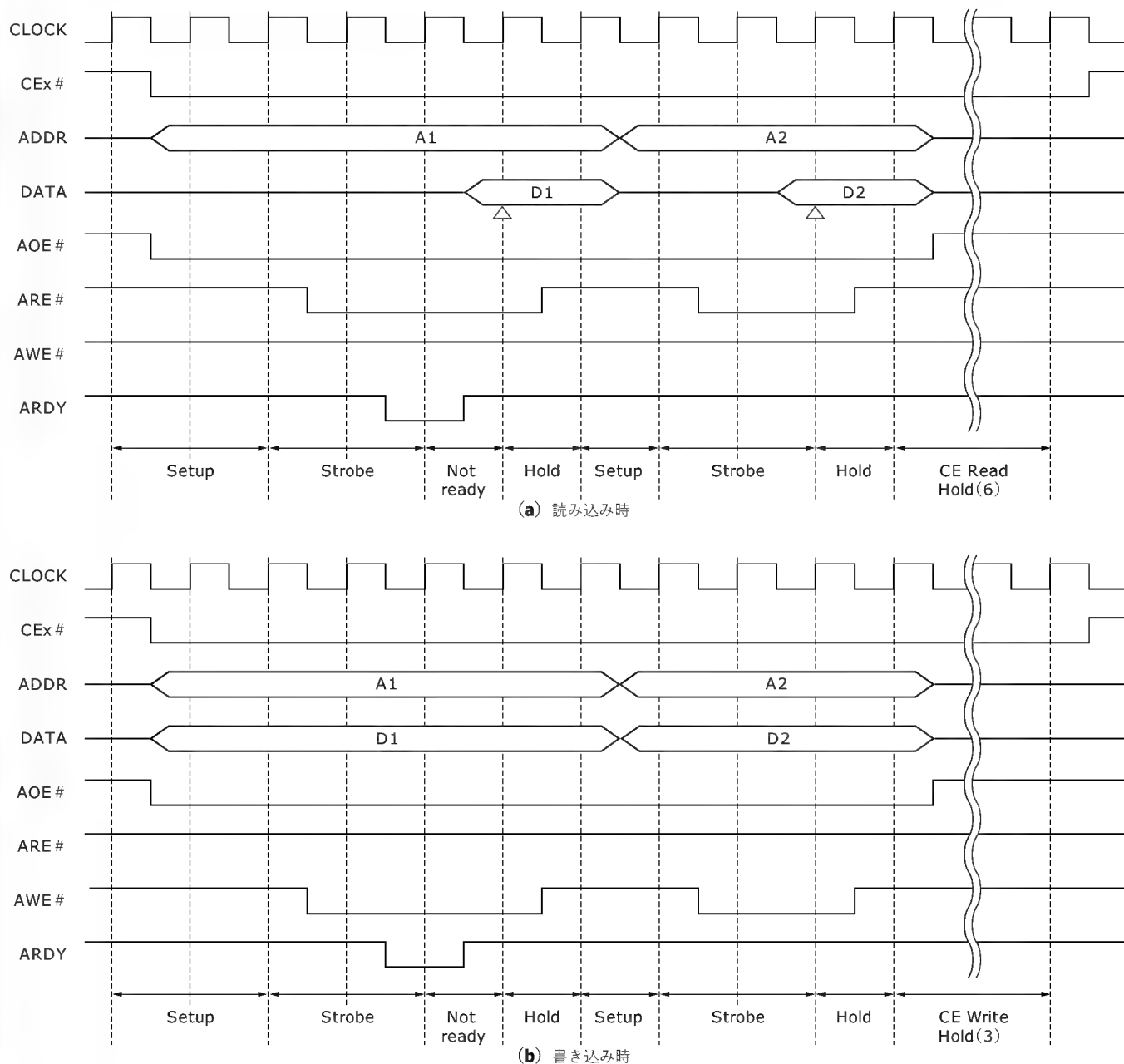
● FPGA/DSPインターフェース

'C6xのEMIFを利用したFPGAとDSPがデータ交換を行うためのモジュールです．EMIFはDSPの外部メモリインターフェースなので，SBSRAM，SDRAM，および非同期SRAMを直接接続することができます．そのため，DSPはそれらのアクセス手順でFPGAと接続されています．つまり，FPGA内部にDSPからのメモリアクセスを受け付ける回路を設計することになります．本稿では，非同期SRAMインターフェースを使用する場合，どのような動作をするかの説明をします．

図6に非同期SRAMをアクセスする場合のタイミングチャートを示します．EMIFが提供する非同期SRAMインターフェースはリクエストアクノリッジ信号を利用するのではなく，リクエスト信号がアクティブになってから，一定のレイテンシ（初期化時にHPI経由で設定）後にアクセスが終了するという手順で行われます．ただし，設定されたレイテンシ以内に処理を終了できない場合は，レディ信号（ARDY）を利用してそのレイテンシを延長することができます．そのため，FPGA内にある他

〔図6〕非同期インターフェースのタイミングチャート

(a) 読み込み時

(b) 書き込み時

のモジュールとの調停を取ることが比較的簡単に実現できます.

● メモリコントローラ

FPGAからDIMMを利用するためのモジュールです．DIMMはリード/ライトアクセスを行う機構以外に，リフレッシュ，プリチャージ，CASレイテンシの設定などを行うモードレジスタセットといった制御アクセス機構を必要とします．DIMMの性能を最大限に利用するためには，バースト長を8またはフルページに設定し，連続アクセスします．利用したいバースト長はアプリケーションによって異なるため，ライブラリではバースト長を自由に変更できるようになっています．さらに，メモリコントローラ内にはFIFOを搭載しているので，連続してデータを取れない場合にもデータは保持されています．なお，このメモリコントローラでは，論理合成時に，利用しない機能が削除されるように設計されています.

● PCIコントローラインターフェース

RYUOH上に搭載されているAMCC社のS5935を制御して，PCなどのほかのPCIデバイスとのデータ転送を実現します．S5935は同期転送，非同期転送の両方を行うことができるため，このモジュールはS5935から提供されるクロックで動作します．FPGAとS5935は32ビットバスで接続されており，データと

アドレスがバスを共有しているので，制御信号にあわせて切り替えられるようになっています．S5935はMailBox，Pass-Thru，FIFOという3種類の通信方式をもっていますが，これらは排他的に実行されるので，ほかのモジュールに対するインターフェースを統一し，そこからすべての機能を利用できるようにしています．

これまで述べた制御機構はすべて独立した構成になっているので，すべてのモジュールを同時に使用することができます．しかし，これらの制御機構は複数のモジュールから同時にアクセス要求が起こることが予想されます．利用状態をユーザーが判断して利用するのは非常にめんどうなので，各制御機構には調停機構とアクセスされる可能性のあるモジュールに対する専用インターフェースを提供しています．また，優先順位は静的に決定していますが，ユーザーがライブラリ利用時に変更できるようになっています．

〔図7〕VisualSpecテンプレート

協調設計環境の構築への要求と方針

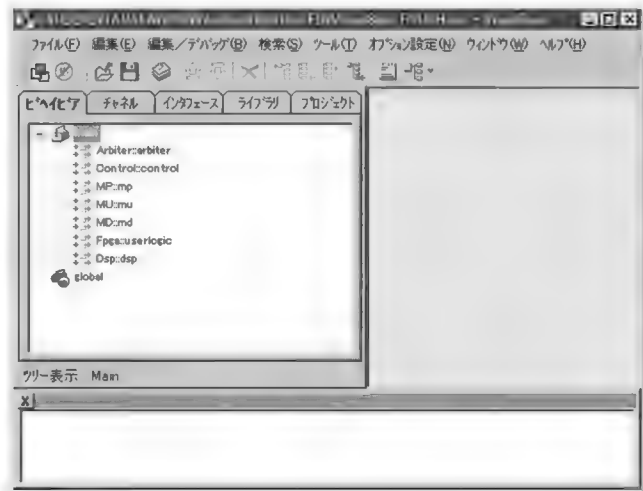
RYUOHを使ってVisualSpecを利用してハードウェア/ソフトウェア協調設計を行ってみました．しかし，RYUOHの開発に必要な機能をすべてVisualSpec上で実現することは，あまり効果的ではないという結論を得ました．そこで，次に挙げる要求にしたがって，RYUOHのためのVisualSpecについて考えてみました．

- ユーザーはメインプログラムだけを記述するようにしたい
- VisualSpecで設計入力した後はツールから生成されるコードに対して手を加えたくない
- 現在のEDAツールの機能を大きく損ないたくない
- ハードウェアとソフトウェアのトレードオフが容易にできるようにしたい

以上を考慮して，協調設計環境開発への方針を決定しました．

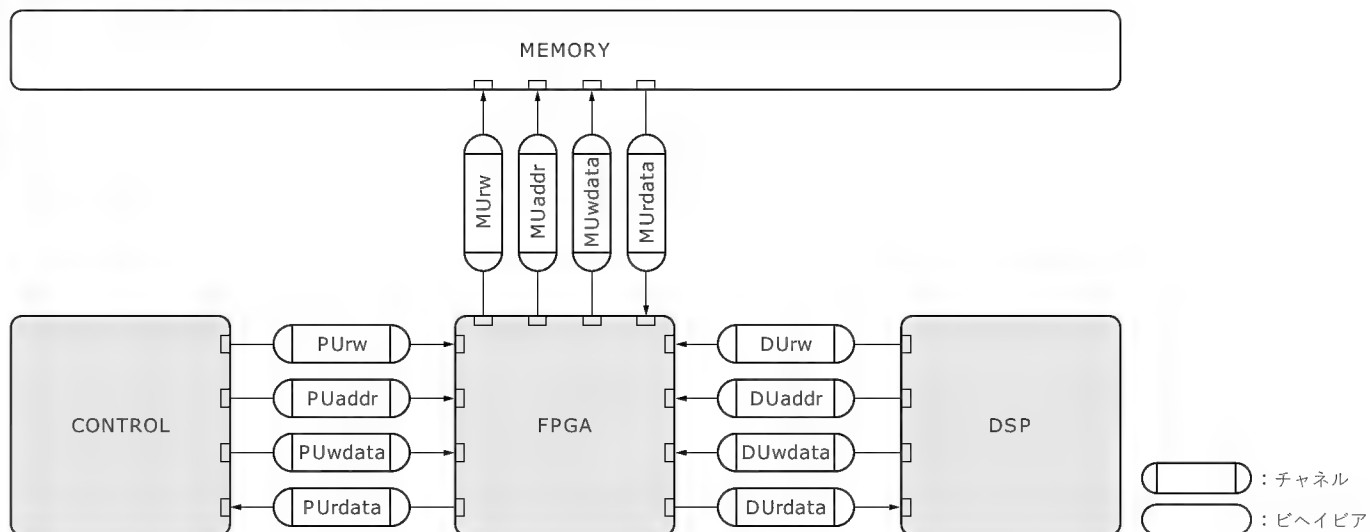
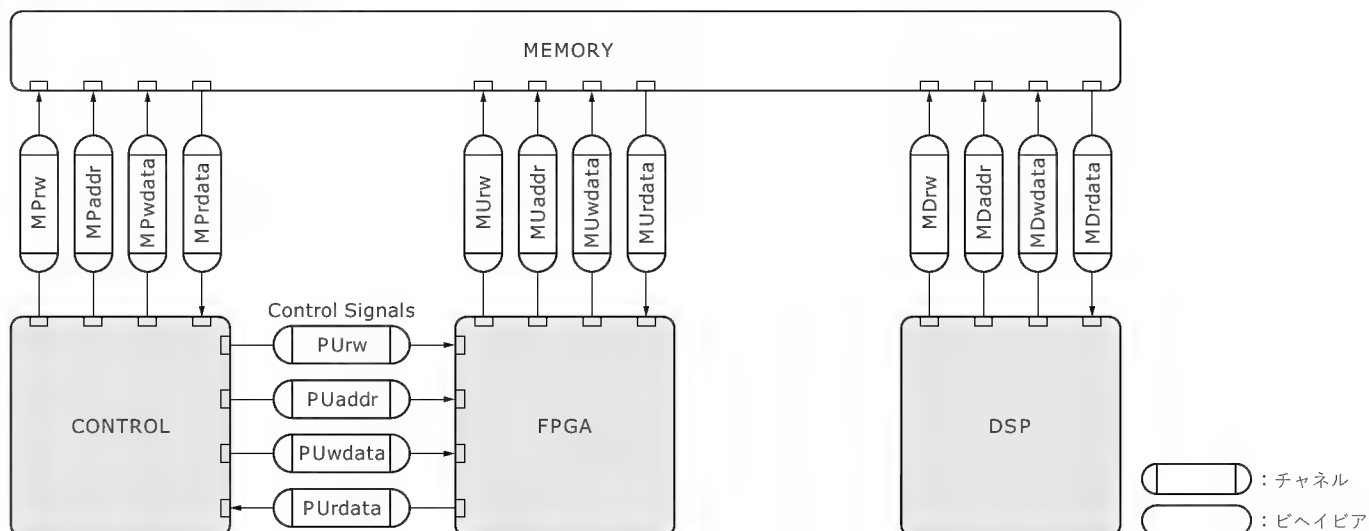
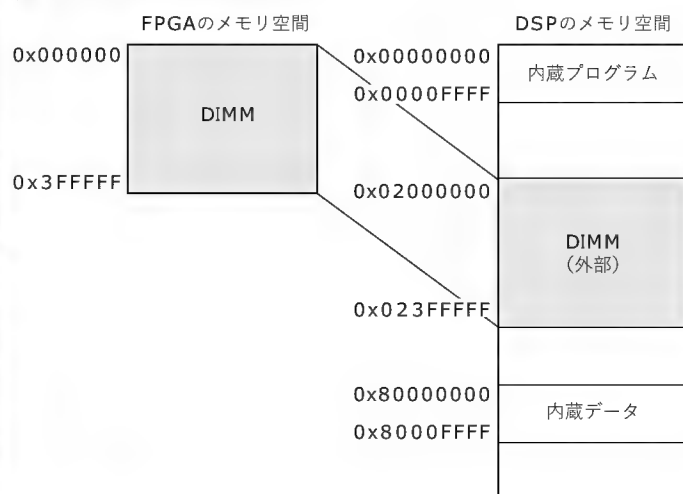
いちばんの問題はRYUOHライブラリSpecCコードで正確に作成し，それらを用いてVisualSpec上でシミュレーションするかどうかという点でした．今回は，シミュレーション時間の増加を防ぐため，VisualSpec上ではライブラリの動作を忠実に記述するのではなく，ライブラリと設計データをネットリストのレベルで接続する際に問題にならない程度のレベルでシミュレーションすることにしました．これを実現するために，RYUOH用のVisualSpecテンプレートを作成しました．

RYUOH向けVisualSpecテンプレート

VisualSpecを利用してRYUOH用のモデルを設計する際，RYUOHに実現できる機能をすべてモデリングすると非常に複雑になります．そこで，RYUOHのユーザーに対して必要最小限の記述だけでモデルの開発を行えるように，RYUOH専用のVisualSpecプロジェクトを作成しました．これにより，ユーザーはこのプロジェクトをVisualSpecで開き，その中にあるFPGA向けユーザーロジックエリア(FPGAビヘイビア)，DSP向けプログラムエリア(DSPビヘイビア)，および制御PC向けプログラム(CONTROLビヘイビア)に記述することで，RYUOH上にモデルを実現できるようになりました(**図7**)．

VisualSpecテンプレートにおける各ビヘイビア間の接続形態はいくつか考えられました．**図8**に示すようなRYUOHの構造を意識したテンプレート構成も候補に挙がりましたが，FPGAビヘイビアとDSPビヘイビアのポート数が大きく異なり，ハードウェアとソフトウェアのトレードオフを容易にできなくなる可能性があるため，**図9**のような構成にしました．この構成もFPGAビヘイビアとDSPビヘイビアのポート数は違いますが，CONTROLビヘイビアとFPGAビヘイビア間の信号線は，設計初期段階ではFPGAの起動要求に使い，処理データはMEMORYビヘイビアにあるという前提で行うので，**図8**のテンプレートよりもトレードオフは比較的容易に行えます．

このテンプレートのビヘイビアには，必要となるコードはすべて記述してあり，ユーザーはその中に動作を記述することでモデルを設計検証することができます．ユーザーが記述するビヘイビアは，FPGAビヘイビア，DSPビヘイビア，およびCONTROLビヘイビアの三つです．CONTROLビヘイビアは，モデル設計後，PCからRYUOHを制御するための制御回路のコードとして利用します．したがって，実際のモデルを記述するビヘイビアはFPGAとDSPとなります．

FPGAビヘイビアとDSPビヘイビアとのSpecC言語コードの違いは，**図10**(p.105)に示す程度です．大きな違いは，FPGAビヘイビアはCONTROLビヘイビアからの信号で起動しますが，DSPビヘイビアではメモリへのポーリングで起動するタイミングを決定しています．また，ポート名なども異なりますが，MUrwとMDrwといったUとDの一文字違いなので，変更(置換作業で簡単に対応可能)することはそれほどたいへんでは

〔図8〕テンプレートの構成(不採用)

〔図9〕テンプレートの構成(採用)

〔図11〕FPGAとDSPのメモリマップ

ありません.

最後に,メモリのオフセットには注意が必要です.RYUOHでは,DIMMが共有メモリとして利用できますが,**図11**に示すように同一データにアクセスする際のアドレスがFPGAとDSPで異なります.FPGAからみた0番地は,DSPでは0x2000000番地になります.つまり,メモリアドレス空間を利用するには,必ずOFFSETといった変数を加算するように記述する必要があります.

ある画像処理アプリケーションの場合

筆者らが用意したVisualSpecテンプレートが容易に利用できるかどうかを検証するために,ある簡単なアプリケーション

〔図10〕FPGAビヘイビアとDSPビヘイビアの違い

```
/******************************************************
* UserLogic.sc                                         *
* Copyright (c) 2003 九州工業大学 マイクロ化総合技術センター *
*                    情報工学部 知能情報工学科          *
******************************************************/

void main( void ) {

  // CONTROLビヘイビアのためのインターフェース
  char  PUrw;
  long  PUaddr, PUwdata, PUrdata;


  // MEMORYLビヘイビアのためのインターフェース
  char  MUrw;
  long  MUaddr, MUwdata, MUrdata;


  // 変数宣言
  short  mode;
  short  width, height;
  unsigned int  DR, DG, DB;
  char  DY, DU, DV;
  char  RY, RU, RV;
  int  i;

  int OFFSET = 0; // FPGA = 0, DSP = 0x2000000


  PUrw = PUrw_r.blockRead();
  if (PUrw == WRITE) {

    // CONTROLからの書き込み
    PUaddr     = PUaddr_r.blockRead();
    PUwdata    = PUwdata_r.blockRead();

  } else {

    // CONTROLからの読み込み
    // 今回は制御を誤ったときのみ対象

    PUaddr   = PUaddr_r.blockRead();
    PUrdata = 0;
    PUrdata_s.blockWrite( PUrdata );

  }


  // FPGAではDSPのようにポーリングは不要であるので,
  // while文は不要. while文の中は, DSPと同等である.
  MUrw = READ;
  MUaddr = MODE_ADDR + OFFSET;
  MUrw_s.blockWrite( MUrw );
  MUaddr_s.blockWrite( MUaddr );
  MUrdata = MUrdata_r.blockRead();
  mode = (short)MUrdata;

        (中略)

}
```

(a) FPGAビヘイビア

```
/*******************************************************
* DSP.sc                                                *
* Copyright (c) 2003 九州工業大学 マイクロ化総合技術センター  *
*                    情報工学部 知能情報工学科           *
*******************************************************/

void main( void ) {

  // MEMORYLビヘイビアのためのインターフェース
  char  MDrw;
  long  MDaddr, MDwdata, MDrdata;


  // 変数宣言
  short mode;
  short width, height;
  unsigned int DR, DG, DB;
  char DY, DU, DV;
  char RY, RU, RV;
  int i;

  int OFFSET = 0x2000000; // FPGA = 0, DSP = 0x2000000

  // このコードは, DSP向けである. DSPではメモリをポーリングし,
  // MUrdataが0から変化するまで動作しない.
  MDrdata = 0;
  while (MDrdata == 0) {
    MDrw = READ;
    MDaddr = MODE_ADDR + OFFSET;
    MDrw_s.blockWrite( MDrw );
    MDaddr_s.blockWrite( MDaddr );
    MDrdata = MDrdata_r.blockRead();
    mode = (int)MDrdata;
  }

      (中略)

}
```

(b) DSPビヘイビア

をモデリングしてみました.

モデリングしたアプリケーションは, ジェスチャ認識を行うための最初の段階の動作をまねたもので, 本学の知能情報工学科の3年生の実験演習でも利用しているものです. 具体的には, CCDカメラの出力画像フォーマットでも用いられるRGB画像から参照したい色を探し出し, その結果を画像で表します. たとえば, **図12**に示すように, ある画像から青色画像を抽出するような作業を行うと, 青い部分だった場所が白く浮き出てくる画像が生成されます.

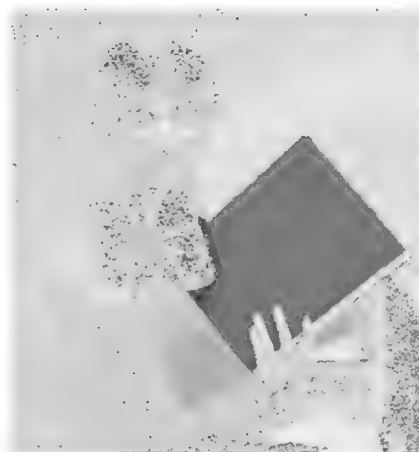
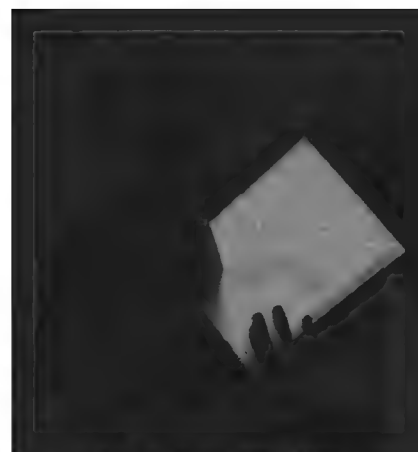
このアプリケーションの動作は, 色空間変換と距離計算の二つに大別できます. 最初の色空間変換では, RGB画像データをYUV画像データに変換します. その変換のための式は, 次の式(1)で表すことができます.

$$\begin{cases} y=(307\times r+604\times g+113\times b)>>10 \\ u=(592\times(b-y))>>10 \\ v=(744\times(r-y))>>10 \end{cases} \quad \cdots\cdots (1)$$

次の距離計算では, YUVデータに変換された参照色との距離計算を行います. その際の式は式(2)で表すことができます.

$$\begin{cases} t_1=(ry-128)-(ty-128) \\ t_2=ru-tu \\ t_3=rv-tv \\ dis=\dfrac{255\times 10}{\sqrt{(t_1)^2+(t_2)^2+(t_3)^2}+10} \end{cases} \quad \cdots\cdots (2)$$

〔図12〕アプリケーションの実行例

(a) 元の画像

(b) YUV画像に変換する

(c) 青い部分を抽出した画像

今回のサンプルはハードウェア/ソフトウェア協調設計の事例なので，上記の二つの処理をハードウェアとソフトウェアのどちらかに割り振るような設計を行ってみます．色空間変換がハードウェアに割り当てられた場合，距離計算はソフトウェアで処理されます．また，逆に色空間変換がソフトウェアに割り当てられた場合，距離計算はハードウェアで処理されます．

● SpecCコード

アプリケーションのSpecCコードの一部を**リスト1〜リスト4**に示します．すべてのリストは本号付属CD-ROM InterGiga No.31に収録しています．このコード群は実際にはVisualSpecから一つのコードとして生成されます．ユーザーが入力するビヘイビアは，FPGA，DSP，およびCONTROLビヘイビアの三つです．今回は，FPGAビヘイビアが二つあるのは，処理を場合分けして入力したためで，DSPビヘイビアのように一つにすることも可能です．また，メモリビヘイビアは，調停機能などを検討していたため四つに分かれています．それらを接続しているトップファイルがRGB2DISになります．

`RGB2DIS.h`：ヘッダファイル
`RGB2DIS.sc`：メインプログラム（**リスト1**）
`FPGA.FD.sc`：FPGA向け色空間変換プログラム（**リスト2**）
`FPGA.DF.sc`：FPGA向け距離計算プログラム（**リスト3**）
`DSP.sc`：DSP向けのコード．色空間変換プログラムと距離計算プログラムが一つになっている（**リスト4**）
`CONTROL.sc`：PC向けの制御コード
`MP.sc`，`MU.sc`，`MD.sc`，`ARBITER.sc`：四つでメモリビヘイビアで構成されている

今回のサンプルコードでは，設計にSpecC固有の記述をあまり利用していないので，C言語設計とほとんど変わりません．メインプログラムは，VisualSpecから生成されるコードなので，`par{ }`などの固有の記述が見られます．

● アプリケーション開発のためのすべての開発フロー

本事例における全体の設計フロー（**図13**，p.111）を紹介しておきましょう．これまで述べてきたように，設計入力には，

〔リスト1〕メインプログラム（RGB2DIS.sc）

```
//
//  VisualSpec_RYUOH.sc
//  2003/06/30 11:57:07 PM
//  Created by VisualSpec 2002 Version1.50
//
#include <sim.h>
#include <vscom.h>

//
// Global variables
//
long     bMemory[MEMSIZE];
long     *Memory;
//
// Interface : IfBr_char
//
interface IfBr_char {
note InterfaceName = "IfBr_char";
note IP = 1;
note VSpecSystem = 1;
note IP_pcl_data_type = "char";
note IP_pcl_read_interface = "Blocking";
    char blockRead(void);
    long canRead(void);
    char read(void);
    long tryRead(char*);
};

//
// Interface : IfBw_char
//
interface IfBw_char {
note InterfaceName = "IfBw_char";
note IP = 1;
note VSpecSystem = 1;
note IP_pcl_data_type = "char";
note IP_pcl_write_interface = "Blocking";
    void blockWrite(char);
    long canWrite(void);
    long tryWrite(char);
```

〔リスト1〕メインプログラム(RGB2DIS.sc)(つづき)

```
};

//
// Interface : IfBr_long
//
interface IfBr_long {
note InterfaceName = "IfBr_long";
note IP = 1;
note VSpecSystem = 1;
note IP_pcl_data_type = "long";
note IP_pcl_read_interface = "Blocking";
    long blockRead(void);
    long canRead(void);
    long read(void);
    long tryRead(long*);
};

//
// Interface : IfBw_long
//
interface IfBw_long {
note InterfaceName = "IfBw_long";
note IP = 1;
note VSpecSystem = 1;
note IP_pcl_data_type = "long";
note IP_pcl_write_interface = "Blocking";
    void blockWrite(long);
    long canWrite(void);
    long tryWrite(long);
};



channel VSpecBrBw_char(void) implements IfBr_char, IfBw_char;
note VSpecBrBw_char.ChannelName = "VSpecBrBw_char";
note VSpecBrBw_char.IP = 1;
note VSpecBrBw_char.VSpecSystem = 1;
note VSpecBrBw_char.IP_pcl_data_type = "char";
note VSpecBrBw_char.IP_pcl_read_interface = "Blocking";
note VSpecBrBw_char.IP_pcl_write_interface = "Blocking";
channel VSpecBrBw_long(void) implements IfBr_long, IfBw_long;
note VSpecBrBw_long.ChannelName = "VSpecBrBw_long";
note VSpecBrBw_long.IP = 1;
note VSpecBrBw_long.VSpecSystem = 1;
note VSpecBrBw_long.IP_pcl_data_type = "long";
note VSpecBrBw_long.IP_pcl_read_interface = "Blocking";
note VSpecBrBw_long.IP_pcl_write_interface = "Blocking";
behavior Arbiter(in char MPreq_r_A, in char MPrw_r_A, in long
 MPaddr_r_A, in long MPwdata_r_A, out long MPrdata_s_A,
 out char MPack_s_A, in char MUreq_r_A, in char MUrw_r_A,
 in long MUaddr_r_A, in long MUwdata_r_A, out long MUrdata_s_A,
 out char MUack_s_A, in char MDreq_r_A, in char MDrw_r_A,
 in long MDaddr_r_A, in long MDwdata_r_A, out long MDrdata_s_A,
 out char MDack_s_A);
behavior Control(IfBw_char MPrw_s, IfBw_long MPaddr_s,
 IfBw_long MPwdata_s, IfBr_long MPrdata_r, IfBw_char PUrw_s,
 IfBw_long PUaddr_s, IfBw_long PUwdata_s, IfBr_long PUrdata_r);
behavior MP(IfBr_char MPrw_r, IfBr_long MPaddr_r,
 IfBr_long MPwdata_r, IfBw_long MPrdata_s, out char MPreq_s_MP,
 out char MPrw_s_MP, out long MPaddr_s_MP, out long
MPwdata_s_MP,
 in long MPrdata_r_MP, in char MPack_r_MP);
behavior MU(IfBr_char MUrw_r, IfBr_long MUaddr_r,
 IfBr_long MUwdata_r, IfBw_long MUrdata_s, out char MUreq_s_MU,
 out char MUrw_s_MU, out long MUaddr_s_MU, out long
MUwdata_s_MU,
 in long MUrdata_r_MU, in char MUack_r_MU);
behavior MD(IfBr_char MDrw_r, IfBr_long MDaddr_r,
 IfBr_long MDwdata_r, IfBw_long MDrdata_s, out char MDreq_s_MD,
 out char MDrw_s_MD, out long MDaddr_s_MD, out long
MDwdata_s_MD,
 in long MDrdata_r_MD, in char MDack_r_MD);
behavior Fpga(IfBw_char MUrw_s, IfBw_long MUaddr_s,
 IfBw_long MUwdata_s, IfBr_long MUrdata_r, IfBr_char PUrw_r,
 IfBr_long PUaddr_r, IfBr_long PUwdata_r, IfBw_long PUrdata_s);
behavior Dsp(IfBw_char MDrw_s, IfBw_long MDaddr_s,
 IfBw_long MDwdata_s, IfBr_long MDrdata_r);
behavior Main(void);

//
// behavior : Main
//
behavior Main(void)
{
note BehaviorName = "Main";
char     MPreq_M = 0;
char     MPrw_M = 0;
long     MPaddr_M = 0;
long     MPwdata_M = 0;
long     MPrdata_M = 0;
char     MPack_M = 0;
char     MUreq_M = 0;
char     MUrw_M = 0;
long     MUaddr_M = 0;
long     MUwdata_M = 0;
long     MUrdata_M = 0;
char     MUack_M = 0;
char     MDreq_M = 0;
char     MDrw_M = 0;
long     MDaddr_M = 0;
long     MDwdata_M = 0;
long     MDrdata_M = 0;
char     MDack_M = 0;
// Child channels
VSpecBrBw_char MPrw();
VSpecBrBw_long MPaddr();
VSpecBrBw_long MPwdata();
VSpecBrBw_long MPrdata();
VSpecBrBw_char PUrw();
VSpecBrBw_long PUaddr();
VSpecBrBw_long PUwdata();
VSpecBrBw_long PUrdata();
VSpecBrBw_char MUrw();
VSpecBrBw_long MUaddr();
VSpecBrBw_long MUwdata();
VSpecBrBw_long MUrdata();
VSpecBrBw_char MDrw();
VSpecBrBw_long MDaddr();
VSpecBrBw_long MDwdata();
VSpecBrBw_long MDrdata();
// Child behaviors
Arbiter arbiter(MPreq_M, MPrw_M, MPaddr_M, MPwdata_M, MPrdata_M,
 MPack_M, MUreq_M, MUrw_M, MUaddr_M, MUwdata_M, MUrdata_M,
 MUack_M, MDreq_M, MDrw_M, MDaddr_M, MDwdata_M, MDrdata_M,
 MDack_M);
Control control(MPrw, MPaddr, MPwdata, MPrdata, PUrw, PUaddr,
 PUwdata, PUrdata);
MP mp(MPrw, MPaddr, MPwdata, MPrdata, MPreq_M, MPrw_M, MPaddr_M,
 MPwdata_M, MPrdata_M, MPack_M);
MU mu(MUrw, MUaddr, MUwdata, MUrdata, MUreq_M, MUrw_M, MUaddr_M,
 MUwdata_M, MUrdata_M, MUack_M);
MD md(MDrw, MDaddr, MDwdata, MDrdata, MDreq_M, MDrw_M, MDaddr_M,
 MDwdata_M, MDrdata_M, MDack_M);
Fpga userlogic(MUrw, MUaddr, MUwdata, MUrdata, PUrw, PUaddr,
 PUwdata, PUrdata);
Dsp dsp(MDrw, MDaddr, MDwdata, MDrdata);
// MAIN
void main(void) {
    // Par-Pipe
    note arbiter.priority = 16;
    note control.priority = 16;
    note mp.priority = 16;
    note mu.priority = 16;
    note md.priority = 16;
    note userlogic.priority = 16;
    note dsp.priority = 16;
    par {
         arbiter.main();
         control.main();
         mp.main();
         mu.main();
         md.main();
         userlogic.main();
         dsp.main();
    }

}
};
```

〔リスト2〕**FPGA向け色空間変換プログラム**(FPGA.FD.sc)

```
//
// behavior : Fpga
//
behavior Fpga(IfBw_char MUrw_s, IfBw_long MUaddr_s,
 IfBw_long MUwdata_s, IfBr_long MUrdata_r, IfBr_char PUrw_r,
 IfBr_long PUaddr_r, IfBr_long PUwdata_r, IfBw_long PUrdata_s)
{
note BehaviorName = "Fpga";
// MAIN
/****************************************************************
* UserLogic.sc                                                  *
* Copyright (c) 2003 九州工業大学 マイクロ化総合技術センター          *
*                               情報工学部 知能情報工学科           *
****************************************************************/

void main( void ) {

  // CONTROLビヘイビアのためのインターフェース
  char  PUrw;
  long  PUaddr, PUwdata, PUrdata;


  // MEMORYLビヘイビアのためのインターフェース
  char  MUrw;
  long  MUaddr, MUwdata, MUrdata;


  // 変数宣言
  short  mode;
  short  width, height;
  unsigned int  DR, DG, DB;
  char  DY, DU, DV;
  char  RY, RU, RV;
  int  i;

  int OFFSET = 0; // FPGA = 0, DSP = 0x2000000


  PUrw = PUrw_r.blockRead();

  if (PUrw == WRITE) {

    // CONTROLからの書き込み
    PUaddr    = PUaddr_r.blockRead();
    PUwdata   = PUwdata_r.blockRead();

  } else {

    // CONTROLからの読み込み
    // 今回は制御を誤ったときのみ対象

    PUaddr  = PUaddr_r.blockRead();
    PUrdata = 0;
    PUrdata_s.blockWrite( PUrdata );

  }


  // FPGAではDSPのようにポーリングは不要であるので,
  // while文は不要. while文の中は, DSPと同等である.
  MUrw = READ;
  MUaddr = MODE_ADDR + OFFSET;
  MUrw_s.blockWrite( MUrw );
  MUaddr_s.blockWrite( MUaddr );
  MUrdata = MUrdata_r.blockRead();
  mode = (short)MUrdata;


  // ファイルサイズを読み込む
  MUaddr = SIZE_ADDR + OFFSET;
  MUrw_s.blockWrite( MUrw );
  MUaddr_s.blockWrite( MUaddr );
  MUrdata = MUrdata_r.blockRead();
  width  = (short)(MUrdata >> 16) & 0xFFFF;
  height = (short)(MUrdata)       & 0xFFFF;

  switch(mode){

  case MODE_FD :

    // 参照データ
    MUaddr = RYUV_ADDR + OFFSET;
    MUrw_s.blockWrite( MUrw );
    MUaddr_s.blockWrite( MUaddr );
    MUrdata = MUrdata_r.blockRead();

    // 処理
    DR = (unsigned char)((MUrdata >> 16) & 0xFF);
    DG = (unsigned char)((MUrdata >>  8) & 0xFF);
    DB = (unsigned char)((MUrdata)       & 0xFF);

    RY = (char)((  307 * DR + 604 * DG + 113 * DB ) >> 10);
    RU = (char)(( -174 * DR - 348 * DG + 522 * DB ) >> 10);
    RV = (char)((  522 * DR - 440 * DG -  82 * DB ) >> 10);

    MUwdata = (long)(((int)RY << 16) & 0xFF0000 |
              ((int)RU <<  8) & 0x00FF00 |
               (int)RV        & 0x0000FF);

    // 書き込み
    MUrw = WRITE;
    MUaddr = RYUV_ADDR + OFFSET;
    MUrw_s.blockWrite( MUrw );
    MUaddr_s.blockWrite( MUaddr );
    MUwdata_s.blockWrite( MUwdata );


    for(i=0; i<width*height; i++){

      // 読み込み
      MUrw = READ;
      MUaddr = RGB_ADDR + OFFSET + i*4;
      MUrw_s.blockWrite( MUrw );
      MUaddr_s.blockWrite( MUaddr );
      MUrdata = MUrdata_r.blockRead();

      // 処理
      DR = (unsigned char)((MUrdata >> 16) & 0xFF);
      DG = (unsigned char)((MUrdata >>  8) & 0xFF);
      DB = (unsigned char)((MUrdata)       & 0xFF);

      DY = (char)((  307 * DR + 604 * DG + 113 * DB ) >> 10);
      DU = (char)(( -174 * DR - 348 * DG + 522 * DB ) >> 10);
      DV = (char)((  522 * DR - 440 * DG -  82 * DB ) >> 10);

      MUwdata = (long)( (( DY << 16 ) & 0xFF0000) |
                (( DU <<  8 ) & 0x00FF00) |
                (( DV       ) & 0x0000FF) );

      // 書き込み
      MUrw = WRITE;
      MUaddr = YUV_ADDR + OFFSET + i*4;
      MUrw_s.blockWrite( MUrw );
      MUaddr_s.blockWrite( MUaddr );
      MUwdata_s.blockWrite( MUwdata );
    }

    // 終了
    MUrw = WRITE;
    MUaddr = FIN_ADDR + OFFSET;
    MUwdata = R2Y_FIN;
    MUrw_s.blockWrite( MUrw );
    MUaddr_s.blockWrite( MUaddr );
    MUwdata_s.blockWrite( MUwdata );

    break;
  }
}
};
```

〔リスト3〕**FPGA向け距離計算プログラム**（FPGA.DF.sc）

```
//
// behavior : Fpga
//
behavior Fpga(IfBw_char MUrw_s, IfBw_long MUaddr_s,
 IfBw_long MUwdata_s, IfBr_long MUrdata_r, IfBr_char PUrw_r,
 IfBr_long PUaddr_r, IfBr_long PUwdata_r, IfBw_long PUrdata_s)
{
note BehaviorName = "Fpga";
// MAIN
/*************************************************************
* UserLogic.sc                                               *
* Copyright (c) 2003 九州工業大学  マイクロ化総合技術センター      *
*                          情報工学部  知能情報工学科            *
*************************************************************/

void main( void ) {

  // CONTROL ビヘイビアのためのインターフェース
  char  PUrw;
  long  PUaddr, PUwdata, PUrdata;


  // MEMORYL ビヘイビアのためのインターフェース
  char  MUrw, MUrw_next;
  long  MUaddr, MUwdata, MUrdata;


  // 変数宣言
  short  mode;
  short  width, height;
  char  DY, DU, DV;
  char  RY, RU, RV;
  char  t1, t2, t3;
  unsigned char  dis;
  int  x, s;
  int  i, j;

  int OFFSET = 0; // FPGA = 0, DSP = 0x2000000



  PUrw = PUrw_r.blockRead();

  if (PUrw == WRITE) {

    // CONTROL からの書き込み
    PUaddr  = PUaddr_r.blockRead();
    PUwdata = PUwdata_r.blockRead();

  } else {

    // CONTROL からの読み込み
    // 今回は制御を誤ったときのみ対象

    PUaddr = PUaddr_r.blockRead();
    PUrdata = 0;
    PUrdata_s.blockWrite( PUrdata );

  }


  // FPGA では DSP のようにポーリングは不要であるので,
  // while 文は不要. while 文の中は, DSP と同等である.
  MUrw = READ;
  MUaddr = MODE_ADDR + OFFSET;
  MUrw_s.blockWrite( MUrw );
  MUaddr_s.blockWrite( MUaddr );
  MUrdata = MUrdata_r.blockRead();
  mode = (short)MUrdata;


  // ファイルサイズを読み込む
  MUaddr = SIZE_ADDR + OFFSET;
  MUrw_s.blockWrite( MUrw );
  MUaddr_s.blockWrite( MUaddr );
  MUrdata = MUrdata_r.blockRead();
  width  = (short)(MUrdata >> 16) & 0xFFFF;
  height = (short)(MUrdata)       & 0xFFFF;

  switch(mode){

  case MODE_DF :

     // データが揃うまで待ち状態
     do{

       MUaddr = FIN_ADDR + OFFSET;
       MUrw_s.blockWrite( MUrw );
       MUaddr_s.blockWrite( MUaddr );
       MUrdata = MUrdata_r.blockRead();

     }while(MUrdata != R2Y_FIN);

     // 参照データ
     MUaddr = RYUV_ADDR + OFFSET;
     MUrw_s.blockWrite( MUrw );
     MUaddr_s.blockWrite( MUaddr );
     MUrdata = MUrdata_r.blockRead();
     RY = (unsigned char)((MUrdata >> 16) & 0xFF);
     RU = (unsigned char)((MUrdata >>  8) & 0xFF);
     RV = (unsigned char)((MUrdata)       & 0xFF);

     for(i=0; i<width*height; i++){

       // 読み込み
       MUrw = READ;
       MUaddr = YUV_ADDR + OFFSET + i*4;
       MUrw_s.blockWrite( MUrw );
       MUaddr_s.blockWrite( MUaddr );
       MUrdata = MUrdata_r.blockRead();
       DY = (unsigned char)((MUrdata >> 16) & 0xFF);
       DU = (unsigned char)((MUrdata >>  8) & 0xFF);
       DV = (unsigned char)((MUrdata)       & 0xFF);

       // 処理
       t1 = (char)((RY-128) - (DY-128));
       t2 = (char)(RU - DU);
       t3 = (char)(RV - DV);

       x = t1 * t1 + t2 * t2 + t3 * t3;
       s = x >> 1;
       for(j=0; j<10; j++){
     if(s == 0)
       break;
     s = (s + (int)(x/s)) >> 1;
       }
       dis = (unsigned char)( 2550 / (s + 10));

       // 書き込み
       MUwdata = (long)dis;
       MUrw = WRITE;
       MUaddr = DIS_ADDR + OFFSET + i*4;
       MUrw_s.blockWrite( MUrw );
       MUaddr_s.blockWrite( MUaddr );
       MUwdata_s.blockWrite( MUwdata );
     }


     // 終了
     MUrw = WRITE;
     MUaddr = FIN_ADDR + OFFSET;
     MUwdata = Y2D_FIN;
     MUrw_s.blockWrite( MUrw );
     MUaddr_s.blockWrite( MUaddr );
     MUwdata_s.blockWrite( MUwdata );

     break;
   }
}
};
```

〔リスト4〕DSP向け色空間変換/距離計算プログラム(DSP.sc)

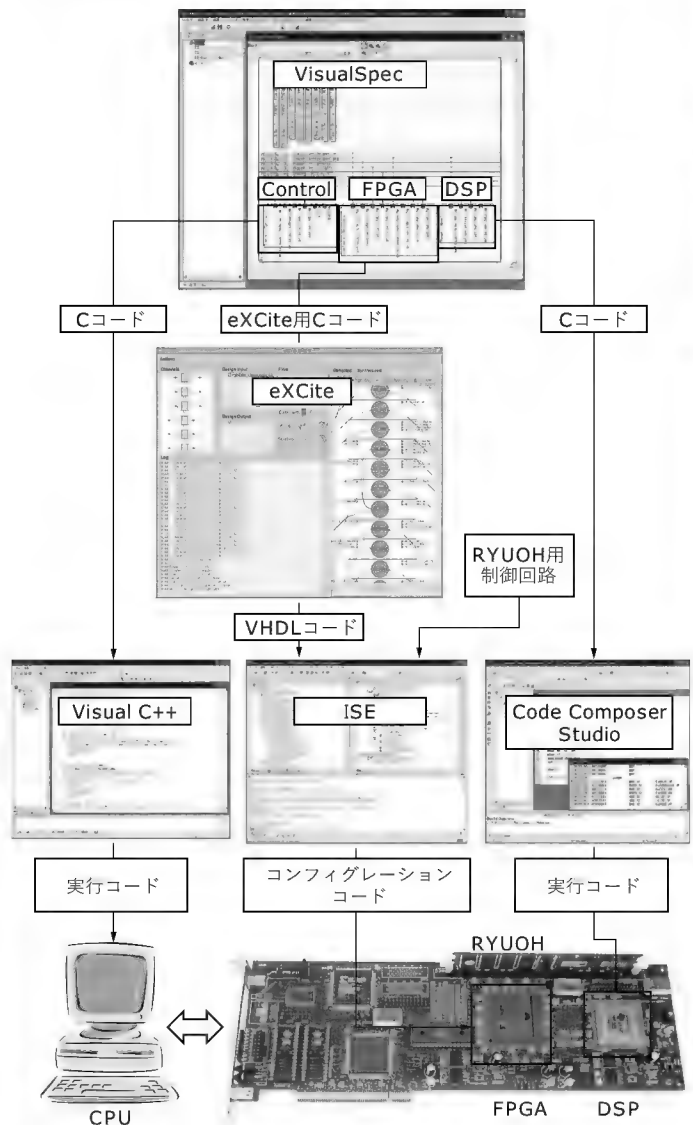
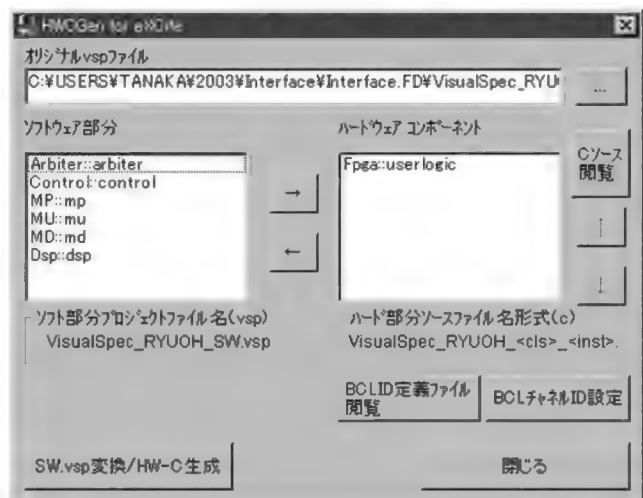
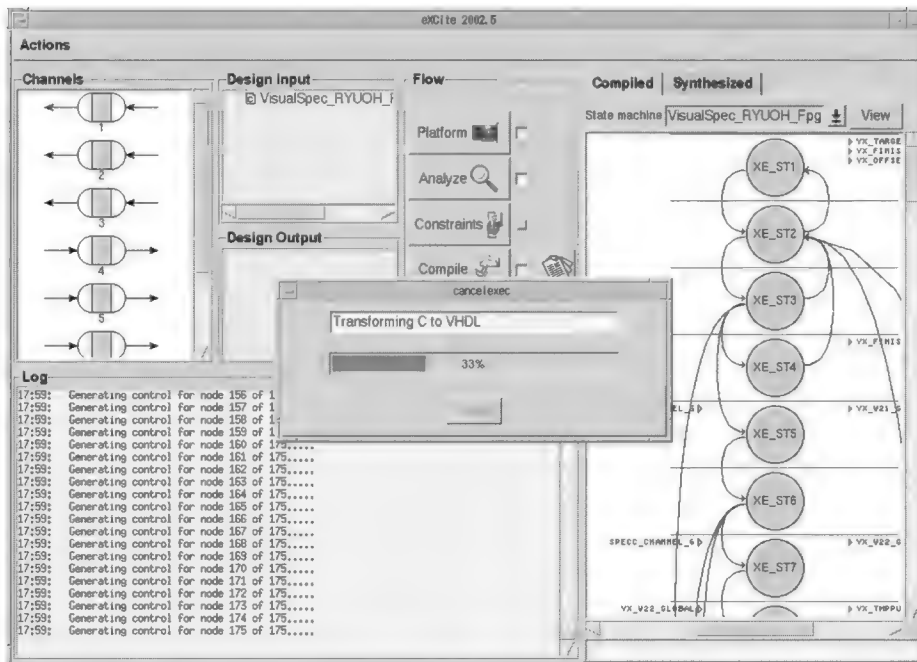
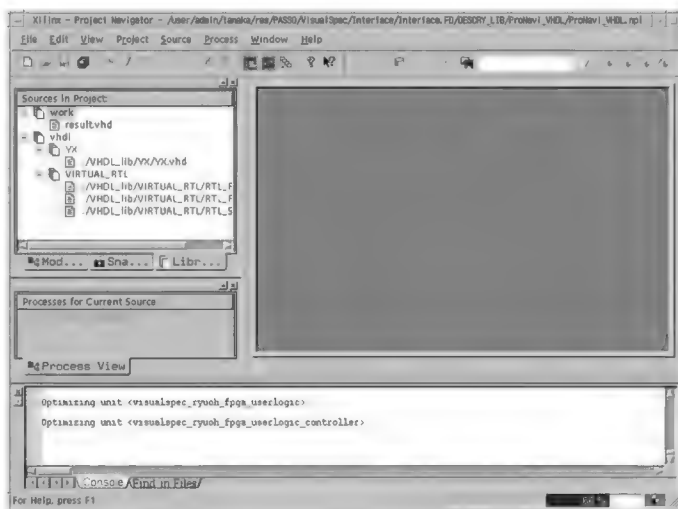
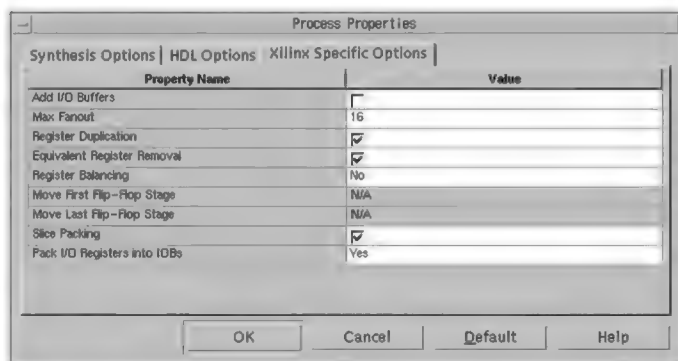
```
//
// behavior : Dsp
//
behavior Dsp(IfBw_char MDrw_s, IfBw_long MDaddr_s, IfBw_long
 MDwdata_s, IfBr_long MDrdata_r)
{
note BehaviorName = "Dsp";
// MAIN
/***************************************************************
* DSP.sc                                                       *
* Copyright (c) 2003 九州工業大学  マイクロ化総合技術センター       *
*                              情報工学部  知能情報工学科          *
***************************************************************/

void main( void ) {

  // MEMORYL ビヘイビアのためのインターフェース
  char  MDrw;
  long  MDaddr, MDwdata, MDrdata;


  // 変数宣言
  short mode;
  short width, height;
  unsigned int DR, DG, DB;
  char DY, DU, DV;
  char RY, RU, RV;
  char t1, t2, t3;
  unsigned char dis;
  int x, s;
  int i, j;

  int OFFSET = 0x2000000; // FPGA = 0, DSP = 0x2000000

  // このコードは，DSP向けである．DSPではメモリをポーリングし，
  // MUrdataが0から変化するまで動作しない．
  MDrdata = 0;
  while (MDrdata == 0) {
    MDrw = READ;
    MDaddr = MODE_ADDR + OFFSET;
    MDrw_s.blockWrite( MDrw );
    MDaddr_s.blockWrite( MDaddr );
    MDrdata = MDrdata_r.blockRead();
    mode = (int)MDrdata;
  }

  // ファイルサイズを読み込む
  MDaddr = SIZE_ADDR + OFFSET;
  MDrw_s.blockWrite( MDrw );
  MDaddr_s.blockWrite( MDaddr );
  MDrdata = MDrdata_r.blockRead();
  width  = (short)(MDrdata >> 16) & 0xFFFF;
  height = (short)(MDrdata)       & 0xFFFF;


  switch(mode){

  case MODE_DF :

    // 参照データ
    MDaddr = RYUV_ADDR + OFFSET;
    MDrw_s.blockWrite( MDrw );
    MDaddr_s.blockWrite( MDaddr );
    MDrdata = MDrdata_r.blockRead();

    // 処理
    DR = (unsigned char)((MDrdata >> 16) & 0xFF);
    DG = (unsigned char)((MDrdata >>  8) & 0xFF);
    DB = (unsigned char)((MDrdata)       & 0xFF);

    RY = (char)((  307 * DR + 604 * DG + 113 * DB ) >> 10);
    RU = (char)(( -174 * DR - 348 * DG + 522 * DB ) >> 10);
    RV = (char)((  522 * DR - 440 * DG -  82 * DB ) >> 10);

    MDwdata = (long)(((int)RY << 16) & 0xFF0000 |
              ((int)RU <<  8) & 0x00FF00 |
              (int)RV         & 0x0000FF);
    // 書き込み
    MDrw = WRITE;
    MDaddr = RYUV_ADDR + OFFSET;
    MDrw_s.blockWrite( MDrw );
    MDaddr_s.blockWrite( MDaddr );
    MDwdata_s.blockWrite( MDwdata );


    for(i=0; i<width*height; i++){

      // 読み込み
      MDrw = READ;
      MDaddr = RGB_ADDR + OFFSET + i*4;
      MDrw_s.blockWrite( MDrw );
      MDaddr_s.blockWrite( MDaddr );
      MDrdata = MDrdata_r.blockRead();

      // 処理
      DR = (unsigned char)((MDrdata >> 16) & 0xFF);
      DG = (unsigned char)((MDrdata >>  8) & 0xFF);
      DB = (unsigned char)((MDrdata)       & 0xFF);

      DY = (char)((  307 * DR + 604 * DG + 113 * DB ) >> 10);
      DU = (char)(( -174 * DR - 348 * DG + 522 * DB ) >> 10);
      DV = (char)((  522 * DR - 440 * DG -  82 * DB ) >> 10);

      MDwdata = (long)( (( DY << 16 ) & 0xFF0000) |
              (( DU <<  8 ) & 0x00FF00) |
              (( DV       ) & 0x0000FF) );

      // 書き込み
      MDrw = WRITE;
      MDaddr = YUV_ADDR + OFFSET + i*4;
      MDrw_s.blockWrite( MDrw );
      MDaddr_s.blockWrite( MDaddr );
      MDwdata_s.blockWrite( MDwdata );
    }

    // 終了
    MDrw = WRITE;
    MDaddr = FIN_ADDR + OFFSET;
    MDwdata = R2Y_FIN;
    MDrw_s.blockWrite( MDrw );
    MDaddr_s.blockWrite( MDaddr );
    MDwdata_s.blockWrite( MDwdata );

    break;

  case MODE_FD :

    // データが揃うまで待ち状態
    do{

      MDaddr = FIN_ADDR + OFFSET;
      MDrw_s.blockWrite( MDrw );
      MDaddr_s.blockWrite( MDaddr );
      MDrdata = MDrdata_r.blockRead();

    }while(MDrdata != R2Y_FIN);

    // 参照データ
    MDaddr = RYUV_ADDR + OFFSET;
    MDrw_s.blockWrite( MDrw );
    MDaddr_s.blockWrite( MDaddr );
    MDrdata = MDrdata_r.blockRead();
    RY = (unsigned char)((MDrdata >> 16) & 0xFF);
    RU = (unsigned char)((MDrdata >>  8) & 0xFF);
    RV = (unsigned char)((MDrdata)       & 0xFF);

    // 処理
    for(i=0; i<width*height; i++){

      // 読み込み
      MDrw = READ;
      MDaddr = YUV_ADDR + OFFSET + i*4;
      MDrw_s.blockWrite( MDrw );
      MDaddr_s.blockWrite( MDaddr );
      MDrdata = MDrdata_r.blockRead();

      // 処理
      DY = (unsigned char)((MDrdata >> 16) & 0xFF);
      DU = (unsigned char)((MDrdata >>  8) & 0xFF);
      DV = (unsigned char)((MDrdata)       & 0xFF);

      t1 = (char)((RY-128) - (DY-128));
      t2 = (char)(RU - DU);
      t3 = (char)(RV - DV);
```

〔リスト4〕DSP向け色空間変換/距離計算プログラム(DSP.sc)(つづき)

```
      x = t1 * t1 + t2 * t2 + t3 * t3;
      s = x >> 1;
      for(j=0; j<10; j++){
    if(s == 0)
      break;
    s = (s + (int)(x/s)) >> 1;
      }
      dis = (unsigned char)( 2550 / (s + 10));

      // 書き込み
      MDwdata = (long)dis;
      MDrw = WRITE;
      MDaddr = DIS_ADDR + OFFSET + i*4;
      MDrw_s.blockWrite( MDrw );
      MDaddr_s.blockWrite( MDaddr );
      MDwdata_s.blockWrite( MDwdata );
    }

    // 終了
    MDrw = WRITE;
    MDaddr = FIN_ADDR + OFFSET;
    MDwdata = Y2D_FIN;
    MDrw_s.blockWrite( MDrw );
    MDaddr_s.blockWrite( MDaddr );
    MDwdata_s.blockWrite( MDwdata );

    break;
  }
}
};
```

〔図14〕VisualSpecのシミュレーション画面

VisualSpecを利用します.シミュレーションは,VisualSpecの機能を利用して行います(**図14**).次に,ハードウェア(FPGA)とソフトウェア(DSP, PC)に分かれて作業が進みます(**図15**).FPGAでは,eXCiteを利用してRTLを生成します(**図16**).さらに,このRTLとRYUOHライブラリを基に,Xilinx社のXST(論理合成ツール)とISE(配置配線ツール)と用いてFPGA用のビットストリームデータを生成します.一方,DSPはTI社のCode Composer Studioを使ってバイナリコードを生成します.また,PC用の制御コードは,Microsoft社のVisual C++ .NETを利用してコンパイルします.次にもう少し詳しく説明をします.

〔図13〕開発フロー

〔図15〕VisualSpecのハードウェア/ソフトウェア分割

〔図16〕YXI eXCite

〔図17〕VHDLライブラリを読み込んだISE

〔図18〕Add I/O Buffersオプション

● 機能シミュレーション

設計後半は，VisualSpecによるシミュレーションで充分検証できますが，設計初期はRTL生成後，HDLレベルの機能シミュレーションを行うことをおすすめします．eXCiteから生成されるRTLはVHDLです．このコードとeXCiteで提供されているライブラリコードを利用してシミュレーションを行えます．ふだん筆者らは，おもにVerilog-HDLを利用していたので，Synopsys社のVCS-MXとCadence Design Systems社のNC-Simといった Verilog-HDLとVHDLの混在シミュレーションが行えるシミュレータを使ってシミュレーションをしました．

● 論理合成と配置配線

論理合成と配置配線は，FPGAベンダであるXilinx社のものを利用します．以前は，論理合成機能は付属していませんでしたが，最近のバージョンではXSTという名称の論理合成ツールが利用できるようになりました．

筆者らの論理合成の工程は二つに分かれます．1度目はeXCiteから生成されたRTLを論理合成します．この際，そのRTLを読み込む以外に，ターゲットとなるFPGAの選択や，eXCiteで用意されているVHDLライブラリを読み込む必要があります(**図17**)．また，“Add I/O Buffers”のチェックをはずすことを忘れないようにしてください(**図18**)．この論理合成では，VisualSpecで設計したモデルのネットリストファイル(ngcファイル)だけを生成することが目的です．2度目の論理合成では，FPGA全体の論理合成を行います(**図19**)．その際，eXCiteから生成されたRTLは含みません．このデータは，次の工程である配置配線の際に2度目の論理合成の結果とマージします．配置配線を行う際は，配置配線を行うネットリストがあるディレクトリ

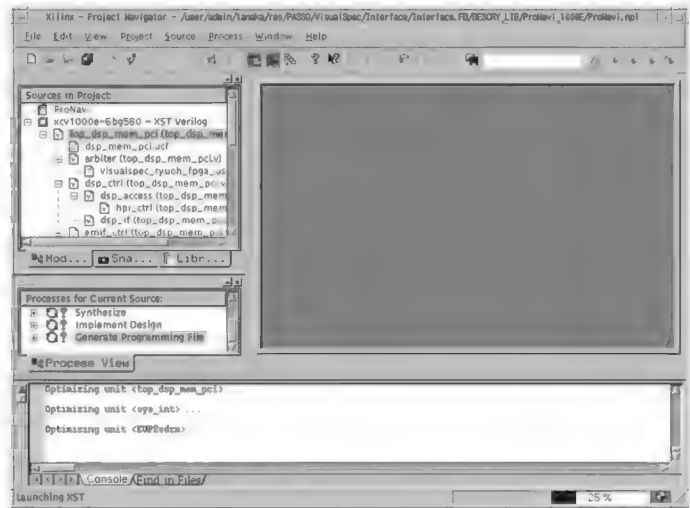
に1度目の論理合成で生成したネットリストファイルをコピーしておきましょう．そうすると，自動的に全体のネットリストにマージされます．配置配線の際の注意すべきオプションはとくにありませんが，配置配線効果のレベルを高くすることで品質の良い実装データができることがあります．

● DSP向けコードのコンパイル

DSP向けのコードは，TI社のCode Composer Studioというコンパイラを利用して次の手順で行います(**図20**)．このコンパイラを利用して，実行用のバイナリファイルを生成し，さらにそのコードをテキストファイルに変更します．このコードをPCからDSPへダウンロードすることで，RYUOH上のDSPは動作を行える状態になります．

このDSP向けコードは，VisualSpecで入力したコードがほぼそのまま利用できます．しかし，**図21**に示すような規則的な変換を行わなければならない箇所があります．現在は，VisualSpecで入力したコードから簡単なperlスクリプトを介して，DSPのコードを生成しています．

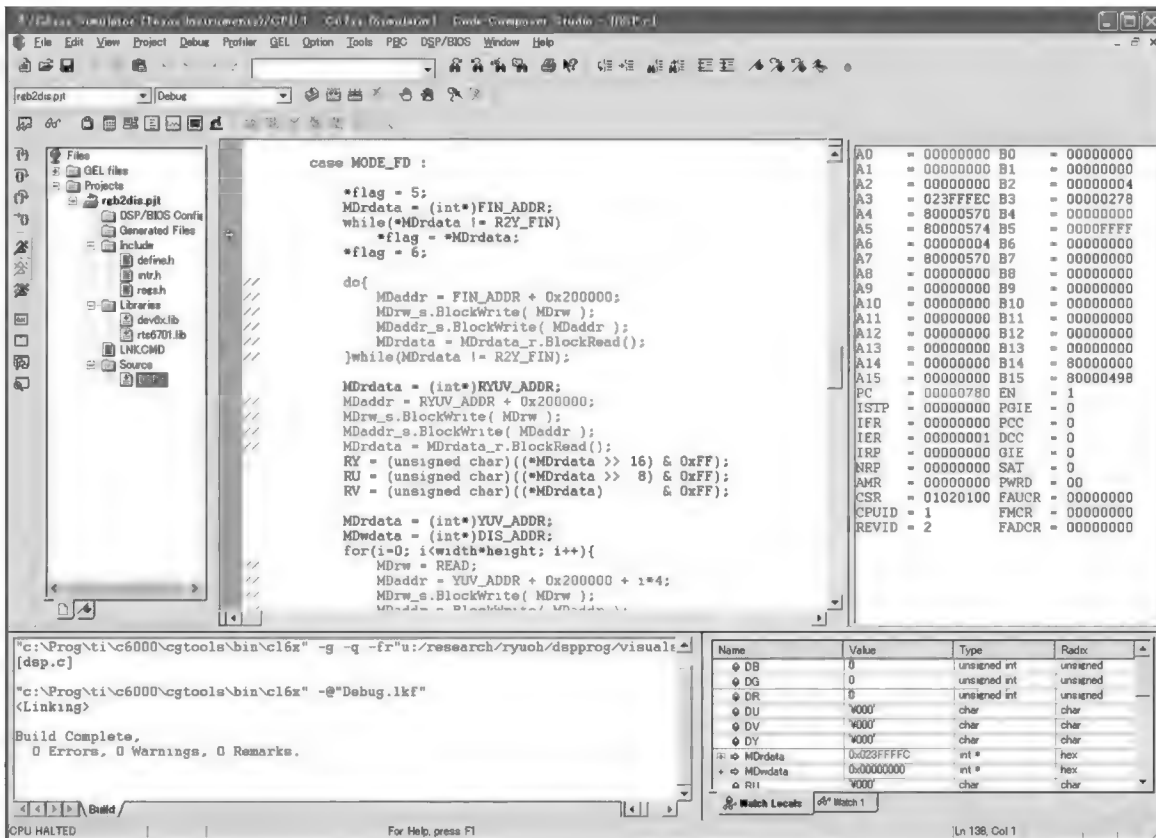
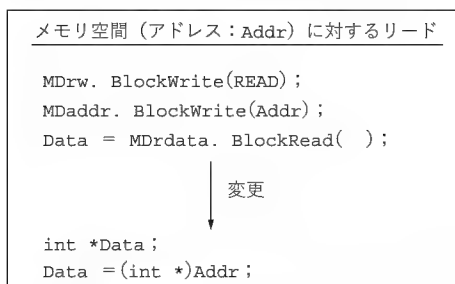
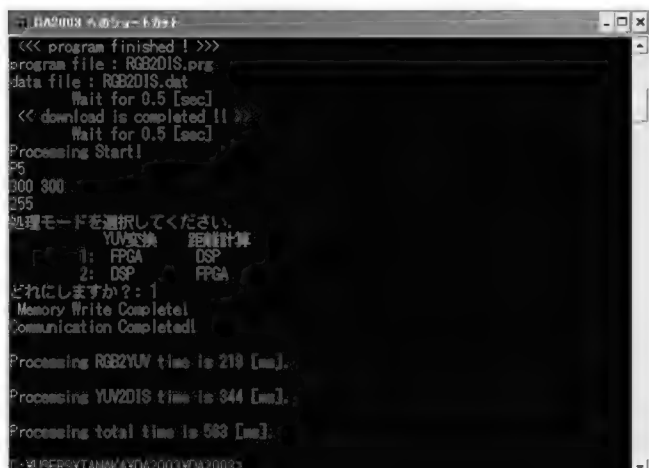
DSPプログラムはコンパイル時(正確にはリンク時)に注意しなければならない点が2点あります．一つはDSPをブートするための割り込み処理ベクタの配置，利用するメモリ空間の指定です．

まず，DSPブートについて説明します．RYUOH上のDSPはHPIからリセット割り込みを発生させることによりDSPがブートします．このとき，割り込み処理ベクタは0番地に配置されている必要があります．この指定にはさまざまなものがありますが，われわれは'C6xのペリフェラルサポートライブラリを使用しています．

次に，前述したRYUOHライブラリを利用する場合の指定方

〔図19〕Xilinx ISE

〔図20〕TI Code Composer Studio

〔図21〕**DSP向けコードのための変換規則**

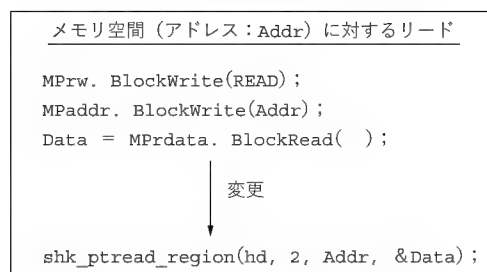
```
メモリ空間（アドレス：Addr）に対するリード

MDrw. BlockWrite(READ);
MDaddr. BlockWrite(Addr);
Data = MDrdata. BlockRead( );

        ↓ 変更

int *Data;
Data =(int *)Addr;
```

〔図22〕**PC向けコードのための変換規則**

〔図23〕**RYUOHの実行画面**（コマンドプロンプト）

法について説明します．DIMM上のメモリを使用する場合は外部メモリ空間：0x02000000～0x023FFFFFを使用しなければなりません．そのため，ユーザーがアクセスしたいアドレスをポインタに代入し，そのポインタを利用することでDIMMを利用します．それ以外の変数はすべてDSPの内部メモリ（プログラム/データ領域，各64Kバイト）を使用します．つまり，コンパイラが指定するメモリはすべて内部メモリであり，FPGAや制御プログラムと共有しているDIMMを使用する場合はユーザーがアドレスを指定します．

● PC向けコードのコンパイル

PC向けのコードは，Microsoft社のVisual C++ .NETコンパイラを利用して次の手順で行います．このPC向けコードも，VisualSpecで入力したコードがほぼそのまま利用できますが，**図22**に示すような規則的な変換を行わなければならない箇所があります．また，RYUOHとアクセスするには，筆者たちの作成したRYUOHドライバを操作するためのAPIを利用する必要があります．

● アプリケーションの実行

すべてのデータの準備が終了すると，アプリケーションの実行が行えます．アプリケーションの実行は，以下の手順で行います．

1）Xilinx社のiMPACTを使ってFPGAにビットストリームデータをダウンロードする
2）DSP用のバイナリデータをメモリイメージのテキストに変換する
3）そのデータをDSP内部のメモリにロードする

以降，制御プログラムから，

4）画像データをDIMMに配置する
5）DSPを起動する
6）FPGAを起動する
7）実行開始

という手順で行います．以上を実行すると，**図23**のように，Windows XPのコマンドプロンプト上に結果が表示され，動作を確認することができます．

おわりに

本章では，SLDLによる設計事例を紹介しました．設計事例に示したように，SLDLを使うことで，FPGAとDSPをある程度意識することなくシステム開発を行うことができます．

組み込みシステムへ要求される高い性能を実現するためには，ハードウェアとソフトウェアのトータルバランスが重要になると予想されます．ハードウェア記述言語とプログラミング言語は似て非なる言語ですから，これらの両方を使いながらシステム設計を行うには，その両方に精通している必要があります．SLDLは，単一言語でシステムすべてを表現できることから，この問題を解決することができる可能性があります．

10年前，筆者は回路図エディタによる回路設計から，HDLによる回路設計に移行しました．当時のHDLは，設計は容易だったものの論理合成ツールの機能が不十分なため，実践の回路設計にはまだまだ利用できないという状況でした．しかし現在では，多くのところで一般的に利用されています．C言語設計の状況は，その当時のHDLに近いものがあります．今後筆者らは，このようなツールが多くの方に利用されることを期待しています．

たなか・こういちろう　九州工業大学

現在注目が
集まっている

SystemCの現状と3.0へのロードマップ

長谷川隆

特集最終章では，今回メインに扱ったシステムレベル記述言語SystemCについて，規格策定過程とその現状，そして新たに登場するSystemC 3.0へのロードマップを解説する．

つい最近まで少なかったSystemC関連の情報だが，ここにきて関連書籍も発売されるなど，徐々に注目されつつある．そこで本章では，SystemCの現状および今後の動向について解説を行う．

（編集部）

1 SystemCとは

SystemCとは，C言語系のシステムレベル記述言語の一つであり，C++言語を基本として，ハードウェアやシステムを記述するのに必要な言語仕様をC++の文法を守った上でクラスライブラリの形で拡張したものである．そしてこのクラスライブラリ（シミュレーションカーネルを含む）は，無償・オープンソースの形で提供されている．SystemCを利用する際には特別な処理系は不要であり，市場に流通している市販の，あるいは無償で配布されているC++コンパイラを用意すれば十分である．したがってSystemCでモデリングやシミュレーションを行うのであれば，非常に安価に開発環境を構築できる．

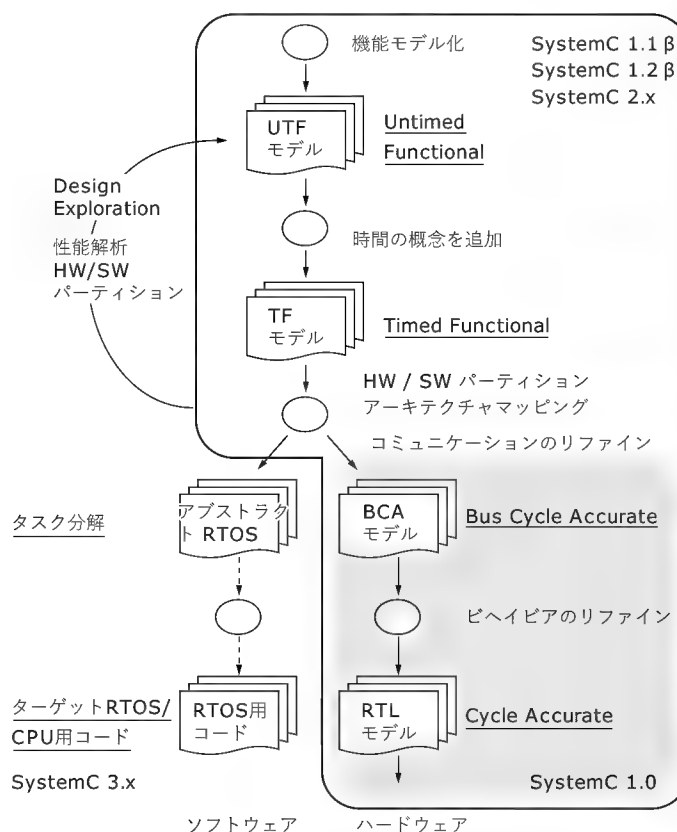
SystemCで記述する対象はハードウェアのみにとどまらず，システムおよびソフトウェアまでも包含する．また，扱える抽象度はレジスタトランスファレベルからシステムレベルと幅広く，異なるレベルが混在した記述とそのシミュレーションも可能である．**図1**に，概略のシステム設計フローに対応させたSystemCの各バージョンがサポートする範囲（一部は計画中のもの）を示す．

2 SystemCの歴史

SystemCの歴史は1990年代後半にさかのぼる．当初はScenicプロジェクトと呼ばれ，現在のSystemCの基本部分（シミュレーションカーネルやクラスライブラリの構成など）の開発が米国Synopsys社と米国カリフォルニア大学アーバイン校との共同で行われた．1997年6月開催のDAC（Design Automation Conference：設計自動化会議）で最初の発表がなされ，その後しばらくの間は限られた数社の間で評価，改良作業が続けられた．1999年後半にオープンソース化する決定がなされ，1999年10月に数多くの賛同を得てSystemCという名前とOSCI（Open SystemC Initiative）の設立が決まり，標準化に向けて動き出した．とはいえ，最初から順風満帆であったわけではない．

● C言語系設計言語乱立時代

SystemCこそが唯一のC言語系設計言語というわけではなく，それ以前にもいくつかのC言語系設計言語が存在していた．しかし，それらの言語は総じて特定のEDAベンダのツー

〔図1〕SystemC各バージョンのサポート範囲

ルを使うための専用言語という位置付けであった．SystemCがそれらと大きく違っていた点は，ツールから独立した言語であり，当初から標準化・流通をめざしたものだったことであろう．

● SystemC 2.0でシステムレベル設計言語に

OSCIが設立された当初のSystemCのバージョンは0.9で，その後しばらくして1.0がリリースされた．しかし，これらのバージョンがサポートする記述範囲はHDL（Verilog-HDL, VHDL）と比べて大差なく，すなわちその名前とは裏腹にシステムレベルで記述する能力が不足していた．このため，2000年夏にランゲージワーキンググループが正式に設置された際の最初の目標は，SystemCを真のシステムレベル記述言語にすることであった．また，シミュレーションカーネルについても新しい構造の提案がなされ，ユーザー定義チャネルを可能にすることも織り込まれた．

そして熱い議論の結果として，まず言語仕様が確定し，2001年2月に公開された．その後の8か月あまりにおよぶ開発作業，検証作業を経て，SystemC 2.0のリファレンスインプリメンテーション（シミュレータ）が2001年10月に正式リリースされた．このバージョンで，ようやく名実ともにシステムレベル記述言語と呼べるようになったといえる．現在のSystemCの最新版は，2002年4月にリリースされた2.0.1である．SystemC 2.xの言語アーキテクチャについて，**図2**に示す．

3 OSCIの活動

OSCI（Open SystemC Initiative）は，SystemCの言語標準の策定および普及を図ることを目的として1999年9月に設立された，非営利法人である（正式に法人化したのは2001年1月）．OSCIは，組織運営のための役員会，技術面の方針決定を行うステアリンググループと各種ワーキンググループから構成されている．

これまでの成果には，SystemC 2.0.1の言語仕様策定とリファレンスシミュレータ，そしてLRMのリリース，SCV（SystemC Verification Standard）のβ版のリリース，各種コンファレンスにおけるSystemC関連イベントの開催あるいは協賛がある．SystemC関連イベントについては年々来場者が増加している．また，SystemCに関する情報発信および議論の場としては，2002年3月よりOSCIの独自管理のWebサイト（http://www.SystemC.org/）を運営しており，SystemCコミュニティの中心的存在となっている．

● 新しい会員クラスが登場

OSCIでは，2002年11月よりそれまでの法人会員，個人会員，名誉会員の3種類のクラスに加えて，準法人会員クラスを追加した．また，合わせて法人会員の年会費を値下げした（$50,000/年→$25,000/年）．

● 難産だった「LRM」

OSCI設立後，SystemCのいくつかのバージョンがリリースされてきたが，その度にLRM（ランゲージリファレンスマニュアル）が準備されていないことが指摘されてきた．もちろん，OSCIもLRMの重要性については認識していたが，開発リソース（＝ボランティア）の不足により，なかなかLRMを作成できないというのが実状であった．

2002年度に入りようやくLRM執筆作業をアウトソーシングながら開始することができ，2003年5月末にはSystemC 2.0.1の言語仕様に即したLRMを一般公開できるところまで漕ぎ着けた．このLRMを元に，年内にもIEEEへの標準化提案が行われる予定である．

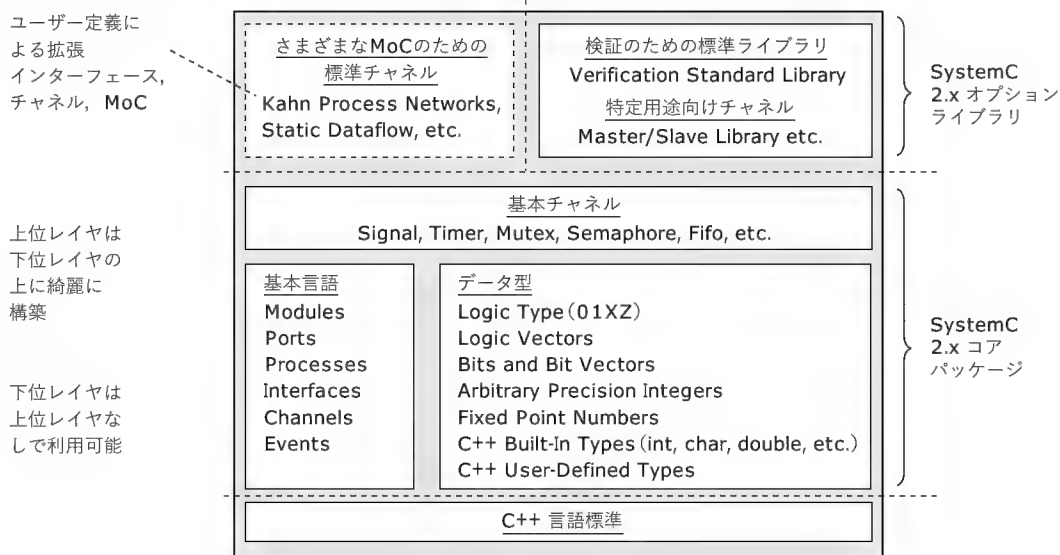
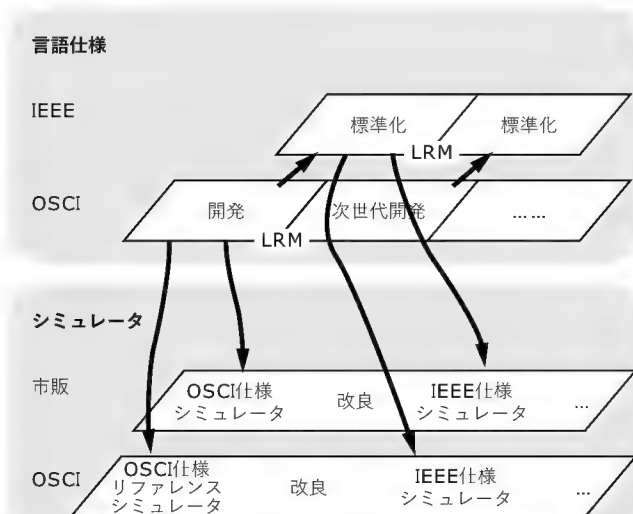
● IEEEへの標準化提案後のOSCI

これまではOSCIが唯一のSystemCに関する管理団体だった

〔図2〕SystemC 2.x言語アーキテクチャ

〔図3〕言語仕様とリファレンスシミュレータ

〔表1〕OSCIにおけるワーキンググループ

ワーキンググループ	活動事項など
ランゲージワーキンググループ	● LRMおよびコアランゲージ部分の改良と新機能のサポート ● 現在はSystemC 2.1およびSystemC 3.0を開発中
検証ワーキンググループ	● 検証方法(Verification Methodorogy)に関して検討，それに必要な言語拡張やコーディングスタイルを標準化
合成ワーキンググループ	● 動作レベル/RTレベルから合成可能なSystemCのサブセットと記述ガイドラインを策定
トランザクションレベルモデリングワーキンググループ	● トランザクションレベルのモデリングガイドラインと，可搬性のためのAPIの策定

が，IEEEにおける標準化がなされると状況が変わってくる．OSCIは言語仕様開発およびリファレンスシミュレータの開発を担ってきたが，IEEEで正式に標準化された暁には，言語仕様そのものはIEEEがオーナーとなる．またIEEEにおける標準化作業の結果，OSCI仕様と若干異なるものが標準化される可能性もゼロではない．したがって，今後の役割分担としては，OSCIは標準化案を策定し，かつそのためのリファレンスシミュレータの開発，IEEEは標準化作業を継続的に行っていくことになる．このあたりの関係を示したのが**図3**である．

ただし，SystemC言語に関する開発作業は年々膨大になってきており，OSCIメンバー会社のリソース(＝これもボランティア)だけでは不足してきている．今後はSystemCコミュニティによる開発活動への参加が期待される．

4 SystemCの今後のロードマップと各種ワーキンググループの活動

SystemC 2.0が正式リリースされた後，SystemCをさらにいろいろな観点から発展・改良するために，いくつかのワーキンググループが作られた．そしてこれらのワーキンググループ全体を統率するのがステアリンググループの仕事となる(各ワーキンググループの概要については**表1**を参照してほしい)．

以下に，各ワーキンググループの活動内容について簡単に述べる．

● ランゲージワーキンググループ

SystemCのコア言語の仕様策定および拡張を担当するワーキンググループである．現在は，ソフトウェアモデリング(アブストラクトRTOS)などを可能とするSystemC 3.0の開発を進めており，要求仕様書をまとめている段階である．このSystemC 3.0によりインストラクションセットシミュレータなどを用いることなく，ソフトウェア，ハードウェアとも抽象度の高いままで性能評価を行えるようになり，確度の高いソフトウェア/ハードウェアの分割やアーキテクチャ選択が可能となる．

SystemC 1.0でハードウェアモデリング，SystemC 2.0でシステムレベルモデリング，SystemC 3.0でソフトウェアモデリングという過程を経て，言語体系的にも完成することになる．なお，SystemC 3.0のリリースに先がけてdynamic thread, fork/joinなどを言語仕様として正式に採用したSystemC 2.1が2003年第3四半期にリリースされる予定である．

図4に，SystemC 2.1，3.0，SCV 1.0の関係を示す．

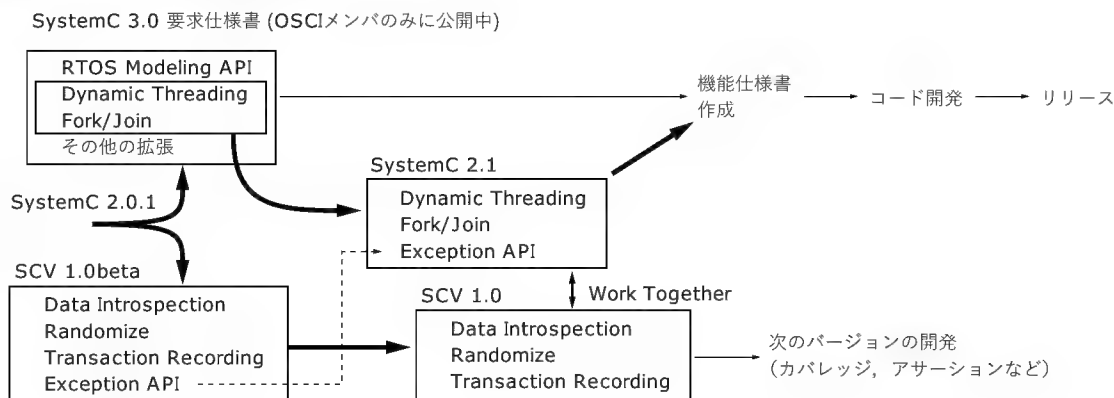
● 検証ワーキンググループ

SystemCを，トランザクションレベルの検証を行うためのフレームワークと位置付け，そのために必要な言語拡張を行うワーキンググループである．現在はVerification Standard 1.0を開発中であり(本稿執筆時点ではβ3版をWebからドキュメントとともに入手可能)，2003年第3四半期には正式版がリリースされる予定である．今回のSCV 1.0版では，トランザクターのモデリングガイド(ドキュメントのみ)，トランザクションレコーディング，ランダム検証のためのランダム生成(制約条件付き，および重み付き)，データの内観(Data Introspection)，といった項目が拡張されている(検証ワーキンググループで拡張された文法についてはSCV_xxxxという予約語が与えられている)．

カバレッジ測定やアサーションについては次の版での実装が予定されている．またフォーマル検証についても今後の課題として議論が進められている．

● 合成ワーキンググループ

2002年10月に設置された比較的新しいワーキンググループであり，SystemCの言語仕様のうち，動作レベルおよびレジスタトランスファレベルのそれぞれについて合成可能なサブセッ

〔図4〕SystemC 3.0，2.1，SCV 1.0の関係図

トを定義し，またモデリングガイドラインを作成することを目標としている．日本企業からの参加がいちばん多いワーキンググループでもある．

● トランザクションレベルモデリングワーキンググループ

2003年3月に設置が決まり，現在活動を始めたばかりのいちばん新しいワーキンググループである．SystemCを用いてトランザクションレベルでモデリングする際に必要な，流通性を考慮したモデリングガイドを策定し，またそれらのトランザクションレベルのモデルを効果的に実装あるいは利用するためのAPIを策定することを目標としている．

5 その他の関連情報

● SystemC関連ツール

2003年のDAC会場では，SystemCを何らかの形でサポートするEDAベンダ，ツールが倍増していた．OSCIのWebページでSystemC関連ツールおよびサービスを紹介しているが，更新が追いつかないほどである．代表的な分野はシミュレータと合成ツールだが，デバッガや文法チェッカといった的を絞ったツールもリリースされている．今のところ合成ツールに関しては決定打と呼べるようなものが出ておらず，今後の改良に期待したい．この分野では国内のベンダも参画しており，ぜひがんばってほしいところである．

シミュレータ関連では，リファレンスシミュレータの速度を改善する目的のものがいくつかリリースされている．また，HDL記述のモデルとSystemC記述のモデルを混在させて協調検証をシームレスに高速に行える環境もいくつかリリースされており，実践的な利用が可能である．

SystemC(および他のC言語系設計言語を含めて)の弱点の一つは，フォーマル検証系のツールが存在していない点である．これが使えるようになると，SystemCの普及がさらに進むと思われる．

6 今後の課題

現時点では，残念ながらまだまだSystemC，あるいはC言語ベース設計について慎重な態度を示す人が多い．しかしすでにC言語系設計言語乱立の時代はすぎ，ほぼSystemCで決まりともいえる状況なので，SystemCを採用することに対して不安な要素はないはずである．

プレスリリースや学会などで実際にSystemCを使用して設計したという発表は，まだまだ少ない．もっとも，SystemCを高位レベルの設計・シミュレーションにだけ用いて，レジスタトランスファレベルは合成せずに従来どおり手設計するような場合には，あえてSystemCを使用したという発表はしていない可能性もある．これは，「SystemCを使用して設計した」という定義を狭く捉えている方が多いということではないだろうか．そもそもSystemCは高位合成のためだけの言語ではないので，SystemCを高位レベルでの設計・検証のフレームワークとして利用するだけで，それは十分にSystemCを活用しているといってよいはずである．

今後は，SystemCを使うべきか否か，という議論よりも，どのように活用するか，という議論が主流になるはずである．それらのキーとなるのが合成ワーキンググループ，検証ワーキンググループ，トランザクションレベルモデリングワーキンググループの成果であるといえる．

以上，SystemCについて，いろいろな側面から概観してみた．SystemCを利用するためのきっかけとしていただければ幸いである．

はせがわ・たかし　富士通(株)/Open SystemC Initiative(OSCI)

フレッシャーズ向け特設記事

Linux/UNIX上でのプログラム開発の主役

GNU開発ツール入門

西田　亙

PC-UNIXが一般的になるとともに，プログラムの開発スタイルも変わりつつある．Windowsが全盛だった時代，開発ツールはWindows上の統合開発環境がすべてだった．しかし，Windowsはエンドユーザーを念頭において整備されているため，プログラマはある面「不遇な」環境下でコーディングせざるを得ない．これに対しUNIXは，古今東西のハッカー達が30年にわたり育て上げてきた「世界最強」のプログラム開発環境である．フリーなPC-UNIXの登場は，すべてのプログラマがこの「夢の開発環境」を無償で享受できることを可能にした．

本稿では，UNIX上のプログラム開発で主役を演じるGNU開発ツールを活用する上で必要となる基礎知識を解説する．

(筆者)

1 プログラム開発環境としてのUNIX

Linuxの台頭により，PC-UNIXはあらゆる分野へ浸透しつつあります．UNIXが官公庁や企業，果ては一般家庭にまで普及することを，10年前に予想していた人はいるのでしょうか？ UNIXの産みの親であるKen Thompson，Dennis Ritchie両氏は，30歳を過ぎた我が子が突如世界のスーパースターとして歓迎されている姿を，複雑な面持ちでながめているのではないかと思います．なぜなら，UNIXは本来プログラマによるプログラマのための開発環境であり，一般ユーザーの使用を念頭に整備されたものではないからです．派手なWindow ManagerとOfficeもどきをUNIXと抱き合わせにし，「無料とオープンソース」を錦の御旗にかかげた「Anti Windows」路線は，彼らの本意ではないでしょう．筆者は，PC-UNIXの普及とともに，その本質が見失われつつあるような気がしてなりません．

そこで本稿では，プログラム開発環境としてのUNIXを今一度見直すための第一歩として，深遠なるGNU開発ツールの世界を紹介します．

統合開発環境の限界

Windows上の統合開発環境は，素っ気ないコマンドラインしかもたないGNU開発ツールとは異なり，プログラマに「優しい」設計になっており，メニューからビルドを選択するだけで，実行ファイルが全自動で出力されます．このような自動運転も良いですが，場合によっては手作業で細かなコード調整を行う必要もあるでしょうし，組み込みシステムのようにROM/RAMを厳密に区別したリンク作業が要求されることもあります．

さらに今後は，開発現場で多種多様なCPUボードを相手にする機会が，ますます増えることと思います．x86もしくはH8さえ扱えればよいという「古き良き時代」は終わりました．当然，開発ツールも対象となるCPUに対応したものをそろえなければなりませんが，Windows環境ではツール一式を新規購入しなければならないケースがほとんどでしょう．

この際，出費がかさむことはもちろんですが，プログラマにとってもっとも深刻な問題は，一から新しいツールの操作方法を習得しなければならない点です．命令インストラクションの勉強に時間を取られるだけならまだしも，ツール環境のマニュアルをひもときながら首っ引きで開発するのでは，たまったものではありません．おそらく今現在も，世界中でこのような悩みを抱えたプログラマが貴重な時間を浪費しているのではないでしょうか．

しかし，どうかご心配なく．UNIX上のGNU開発ツールは，上に述べた問題をことごとく解消してくれることでしょう．ただし，この最強の開発ツールにも弱点が存在します．それは，「プロフェッショナル」のためのツールであることに由来する共通した問題ですが，使用する前に道具の特性を熟知しておく必要がある点です．このため，GNU開発ツールが万人向けの道具とは言い難いのも事実です．しかし，ひとたびその操作に通じることができれば，これほど頼りになる開発ツールはほかにないでしょう．本稿をきっかけとして，一人でも多くの方々にGNU開発ツールの素晴らしさを体験していただければ幸いです．

3 Hello, world!の正体

まず最初に「開発ツールとは何か？」について考えてみましょう．じつは，この問題に答えるためには「実行ファイル」に関する正確な知識が必要となります．DOS時代，`.COM`型式のプログラムはオフセット`0x100`番地から始まる，単純なバイナリ実行コードでした．若かりし頃，ソフトウェア割り込みである

INT 21hを呼び出し，ディスク入出力処理などを行っていたことを懐かしく思い出される方もおられることでしょう．後で解説しますが，PC-UNIX上のプログラムも基本的にはDOSと同じしくみを採用しています．時代や環境は変わっても，基本が変わることはないのです．

その後.COMが.EXE型式に取って代わられたように，実行ファイル形式もまたOSとともに進化を重ねてきました．現在のUNIX上の代表的な実行ファイル形式は，Executable and Linking Format(ELF)と呼ばれるものです．その詳細について今回は割愛しますが，実行コードにラベルや変数などのシンボル情報やデバッグ情報が付加されたものです．この冗長性によりファイルサイズが肥大化している点に注意してください．

それでは，プログラムの正体を見きわめてみましょう．まず，一般的なHello, world!をUNIX上で作成してみます．通常であれば，有名なhello.cが登場するところですが，次のようなアセンブリソースを用意してください(hello-code.s，**リスト1**)．

いきなりのアセンブリソースに面食らった方も多いかと思いますが，これはhello.cの実行コード部分をLinuxカーネル用にx86アセンブリ言語で書き下ろしたものです(#以降はコメント)．“Hello, world!”メッセージを格納するデータ領域(C言語で表現すればchar msg[])は，意図的に別のファイルhello-msg.s(**リスト2**)で定義してみました．

アセンブリソース中でのラベルはC言語と同じようにコロンを後置して定義しますが，このように定義されたラベルはデフォルトではローカルシンボルとして扱われます(C言語のstatic宣言に相当)．このため，外部オブジェクトファイルから参照するためには，明示的に.global指示子を用いてグローバル宣言する必要があります．.global _start，.global msgにより，二つのシンボルはグローバル化され，hello-code.o中からmsgデータを参照可能となります(.extern msg宣言により，msgは外部ファイルに存在することを明示)．_startはldがプログラムの開始アドレスとして認識するシンボルです(C言語のmainに相当)．アセンブリ言語では，C言語の型や構造体のような高級な概念は存在せず，データ領域もしくはコードの開始アドレスを示す「シンボル」がすべてを決定します．このためGNU開発ツールでは，シンボル関連のツールがたいへん充実しています．

hello-code.s中の.textおよびhello-msg.s中の.data指示子は，対象となるセクションを指定するためのものです．詳細は後述しますが，この宣言により実行コードは.textセクションに，メッセージデータは.dataセクションに格納されることになります．

次に，実行コードを見てみましょう．UNIXでは「システムコール」を通じて，ユーザープログラムがカーネルにさまざまな処理を依頼しますが，Linuxの場合その実体はソフトウェア割り込み128番(int 0x80)です．hello-code.s中では二つのシステムコールを利用していますが(9，13行)，一つはwriteシステムコールであり，標準出力(ファイルディスクリプタ1番)へ，msg領域に格納された文字列を出力します．この結果，コンソールにベル音(¥007)をともなったHello, world!が表示されるはずです．二つめはプロセスを終了するためのexitシステムコールであり，終了ステータスコードは親プロセスであるシェルに返されます．

なお，皆さんになじみの深いprintf関数がこのソース中には記述されていない点に注目してください．printfは標準ライブラリ関数ですが，その内部ではこのようにwriteシステムコールを呼び出しているのです．本プログラムの細部を理解する必要はありませんが，UNIX上のプログラムの本質がシステムコールにある点を理解することは，たいへん重要です．それでは，このソースリストを実行可能ファイルに生まれ変わらせてみましょう．

アセンブル

アセンブリソースに命を吹き込むためには，二つの作業「アセンブル」および「リンク」が必要になります．アセンブルとは，「アセンブリ言語を機械語に翻訳」するための作業であり，アセ

〔**リスト1**〕**hello-code.s**

```
1  .extern msg                      # char msg[] is an external symbol
2  .text                            # select .text section
3  .global _start                   # declare _start as a global symbol
4  _start:                          # ld expects "_start" as an entry address
5          movl    $4, %eax         # write (fd, buf, count) system call
6          movl    $1, %ebx         # fd = STDOUT
7          movl    $msg, %ecx       # buf = msg
8          movl    $15, %edx        # count = strlen(msg)
9          int     $0x80            # switch to kernel
10
11         movl    $1, %eax         # exit(status) system call
12         movl    $123, %ebx       # status = 123
13         int     $0x80            # switch to kernel
```

〔**リスト2**〕**hello-msg.s**

```
1  .data                            # select .data section
2  .global msg                      # declare msg as a global symbol
3  msg:                             # char msg[] = "..."
4          .string "Hello, world!¥007¥n"
```

〔図1〕アセンブル

```
$ as -o hello-code.o hello-code.s
$ as -o hello-msg.o hello-msg.s
$ wc hello-*.o
      0       5     572 hello-code.o
      1       4     477 hello-msg.o
      1       9    1049 total
$ file hello-code.o hello-msg.o
hello-code.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
hello-msg.o:  ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

〔図2〕逆アセンブル表示

```
$ objdump -D hello-code.o

hello-code.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0:   b8 04 00 00 00          mov    $0x4,%eax
   5:   bb 01 00 00 00          mov    $0x1,%ebx
   a:   b9 00 00 00 00          mov    $0x0,%ecx
   f:   ba 0f 00 00 00          mov    $0xf,%edx
  14:   cd 80                   int    $0x80
  16:   b8 01 00 00 00          mov    $0x1,%eax
  1b:   bb 7b 00 00 00          mov    $0x7b,%ebx
  20:   cd 80                   int    $0x80
Disassembly of section .data:
```

〔図3〕.dataセクションの内容をダンプ

```
$ objdump -j .data -s hello-msg.o

hello-msg.o:     file format elf32-i386

Contents of section .data:
 0000 48656c6c 6f2c2077 6f726c64 21070a00  Hello, world!...
```

〔図4〕sizeコマンドの実行結果

```
$ size hello-code.o hello-msg.o
   text    data     bss     dec     hex filename
     34       0       0      34      22 hello-code.o
      0      16       0      16      10 hello-msg.o
```

ンブリソース中に記述された一行一行をx86の機械語に置き換えます．GNU開発ツール中において，アセンブル作業はas(ASsembler)が担当します．引き数としてアセンブリソースファイルを指定し，-o(Output)オプションで出力オブジェクトファイル名を指定します．

なお，-oオプションを省略するとa.out(Assembler OUTput)と名付けられたオブジェクトファイルが出力されますが，この慣習はその昔アセンブラが実行ファイルを直接生成していた歴史に基づいているようです．それでは，さっそくアセンブルしてみましょう(**図1**)．

アセンブルが終了すると，hello-code.oおよびhello-msg.oという，それぞれ572,477バイトのファイルが作成されました(以後，出力ファイルのサイズは利用環境によって異なることがある)．fileコマンドでその正体を確かめると，ELF型式であることがわかります(表記がexecutableではなく，relocatableになっている点に注目)．このファイル中に，翻訳された機械語やメッセージが格納されているはずですが，残念ながらバイナリファイルであるため，その中身は16進数の羅列にすぎません．

こんなときは，GNU開発ツールに含まれるobjdump(OBJect file DUMP)を活用します．objdumpに-D(Disassemble)オプションを付けると，指定されたオブジェクトファイルの中身が逆アセンブル表示されます(**図2**)．

ソース中のmovl $4, %eaxという一行は，B8 04 00 00 00という5バイトに変換されていることがわかります．B8はレジスタへの即値転送処理命令コードであり，続く4バイトはリトルエンディアンで表現した4です．その他のアセンブリソース行も，x86機械語に逐一翻訳されていることがわかります．これが，最近ではすっかり見かけることの少なくなった「アセンブル」の結果です．Cコンパイラを縦横無尽に使いこなすためには必須の知識ですから，この機会に理解を深めておきましょう．

hello-code.oには実行コードが格納されていますが，hello-msg.oには文字列データしか定義されていないので，逆アセンブルでは困ります．このような場合のために，objdumpには-jおよび-sオプションが用意されています．-sは-jで指定されたセクション(後述)の内容をダンプするためのオプションです．今回は-j .data -s (.dataセクションの内容をダンプ)としてください(**図3**)．

たしかにhello-msg.o中に“Hello, world!”が.dataセクション中にベル(0x07)，改行(0x0A)，およびヌルターミネータ(0x00)を含めて配置されていることが確認できました．このように，GNU開発ツールではアセンブラ/リンカ/Cコンパイラばかりでなく，高機能な周辺ツール群があらゆる面からプログラマをサポートし，効率的な開発および詳細なコードの検証を可能にしています．

5 セクション構造

さて，これらの結果によるとhello-code.oの実行コードは34バイト，hello-msg.o内部のメッセージデータは16バイトですから，ファイルサイズは400～500バイト以上肥大していることになります．この冗長性は何に由来しているのでしょうか？　GNU開発ツール中のsizeおよびreadelfコマ

ンドで，その秘密を探ってみましょう(**図4**，前頁)．

sizeは，ファイル内部に存在する三つの「セクション」のサイズを表示するためのコマンドです．この結果によれば，hello-code.o中ではtextという名前のセクションのサイズがちょうど34バイト，hello-msg.o中ではdataが16バイトになっていることがわかります(text/data/bssはsizeコマンドの慣習的な表記方法であり，GNU開発ツールでは.text/.data/.bssと表記される)．このように，UNIX上の実行ファイルは複数のセクションから構成されており，実行コードやデータはそれぞれ対応するセクションに格納されます(**表1**)．

アセンブラが出力したオブジェクトファイルを，シンボルとセクションの関係から解析してみましょう．ELFオブジェクトファイル中に含まれるシンボルを解析するためには，nm(NaMe)コマンドを使います(**図5**)．

真ん中のフィールドには，一文字の略語で各シンボルが属するセクションが表示されています．hello-code.o中では_startから始まる実行コードが.text(T)セクションに，hello-msg.o中ではmsgから格納されているメッセージデータが.data(D)セクションに割り当てられています．このセクション略文字が大文字の場合はグローバルシンボル，小文字の場合はローカルシンボルであることを意味しているので，二つのソースファイル中で指定した.global指示子は的確に動作していることがわかります．

また，hello-code.o中でmsgの所属セクションがU(Undefined)になっていますが，これは同ソースファイル中でmsgが.extern宣言されたためです．このように，nmコマンドはプログラムのビルド過程を把握するためにはたいへん役に立つツールなので，積極的に活用することをお勧めします．

さて，せっかくの機会ですから，Cソース中のコードと各種変数が上記セクションに割り当てられるようすを自分の目で確認しておきましょう．次のようなsections.c(**リスト3**)を用意してください．

プログラム内容は簡単なもので，初期化済み変数data，非初期化変数bss，定数変数rodata，関数functionの四つを定義しています．このソースリストをコンパイルして，オブジェクトファイルに変換します(**図6**)．

後で説明しますが，Cソースのコンパイルにはgccコマンドが使われます．デフォルトでは実行ファイルを生成するので，今回は-c(Compile)オプションを指定し，コンパイルのみを行います．あわせて，-fno-commonというオプションを指定していますが，これは非初期化変数を.bssセクションに割り当てるためのものです(デフォルトでは.commonセクションに配置される)．

コンパイルが終わると，生成されたオブジェクトファイルsections.o中に存在するシンボル一覧をnmコマンドを用いて表示します．-g(Global)はグローバルシンボルのみを表示するためのオプションです．たしかに，bss変数は.bssセクション(B)に，data変数は.dataセクション(D)に，function関数は.textセクション(T)に，rodata変数は.rodataセクション(R)に配置されていることがわかります．

このように，Cコンパイラが出力するオブジェクトファイル中のコードや変数は，これら四ついずれかのセクションに属することになります．通常のアプリケーションを作成する場合はセクションの存在を意識する必要はないものの，ROM化/割り込みテーブルの作成/ターゲット機器のメモリマップに合わせたコード/データ割り当てなどを行う場合は，リンカスクリプトを用いてセクションの配置方法を細かく制御しなければなりません．

6 ELFの冗長性

話題を元に戻しましょう．hello-code.o，hello-msg.oのファイルサイズが，400バイト以上も水増しされている原因はどこにあるのでしょうか．こんなときは，readelfコマンドを用いて，ELFファイルの内部を詳細に解析します(**図7**)．

〔図5〕ELFオブジェクトファイル中に含まれるシンボルの解析

```
$ nm hello-code.o hello-msg.o

hello-code.o:
00000000 T _start
         U msg

hello-msg.o:
00000000 D msg
```

〔図6〕sections.cをコンパイルする

```
$ gcc -c -fno-common sections.c
$ nm -g sections.o
00000000 B bss
00000000 D data
00000000 T function
00000000 R rodata
```

〔表1〕セクション

セクション名	内　容	nmコマンド中の表記
.text	実行コード領域	T
.data	初期化済みデータ領域	D
.rodata	固定データ領域(参照のみ可能，変更不可)	R
.bss	非初期化データ領域	B

〔リスト3〕sections.c

```
1  int data = 123;
2  int bss;
3  const int rodata = 456;
4
5  int function() {
6    return 789;
7   }
```

〔図7〕readelf コマンドの実行

```
$ readelf -S hello-code.o
There are 8 section headers, starting at offset 0x88:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000022 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000234 000008 08      6   1  4
  [ 3] .data             PROGBITS        00000000 000058 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000058 000000 00  WA  0   0  4
  [ 5] .shstrtab         STRTAB          00000000 000058 000030 00      0   0  1
  [ 6] .symtab           SYMTAB          00000000 0001c8 000060 10      7   4  4
  [ 7] .strtab           STRTAB          00000000 000228 00000c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

〔図8〕リンク

```
$ ld -o helloasm hello-code.o hello-msg.o
$ wc -c helloasm
    720 helloasm
$ file helloasm
helloasm: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
$ ./helloasm ; echo $?
Hello, world!
123
```

-S(Sections)は，ファイルに含まれるすべてのセクション情報を表示するためのオプションです．sizeは，プログラム実行のために必要な一部のセクション情報しか表示しておらず，実際には.shstrtab/.symtab/.strtabなど余分なセクションが存在していることがわかります．サイズフィールドを見ると，.textセクションが34バイト，後者の三つは48，96，12バイトとなっています．このほかにもELFヘッダやセクション情報を必要とするために，500バイトを超える余分な情報が付加されているわけです．

リンク

アセンブルにより機械語への翻訳が完了すると，リンク作業によりコードとデータの最終的な配置アドレスを決定します．GNU開発ツールではld(linker and LoaDer)コマンドがリンクを担当します．実際にリンクしてみましょう(**図8**)．

-o(Output)は出力ファイル名を指定するためのオプションですが，二つのオブジェクトファイルからhelloasm実行ファイルを生成するよう指示しています．このように，引き数には複数のオブジェクトファイルやライブラリが並びます．

リンク作業が終了すると，720バイトの実行ファイルができあがりました．fileコマンドでその内容をチェックすると，今度はELF executableとなっていることがわかります．さっそく実行してみると，意図どおりメッセージが表示され，シェルには123が返されています($?はプロセス終了時のステータスコードを格納するシェル変数)．ここで，再度helloasmの逆アセンブルを行ってみましょう(**図9**)．

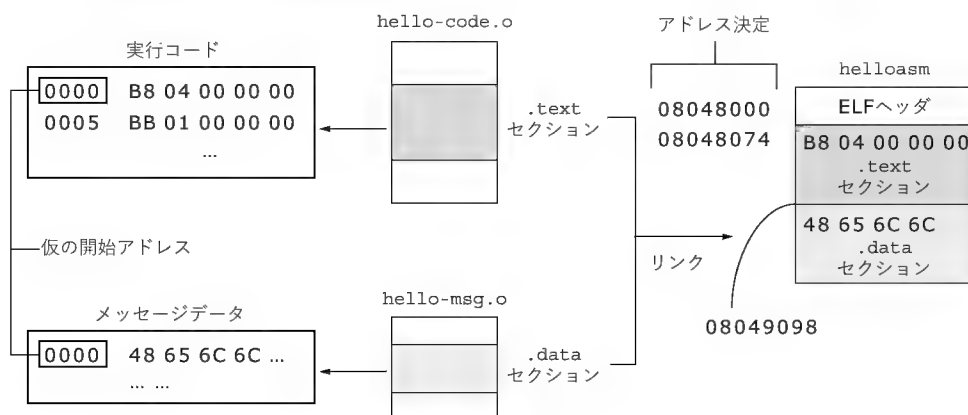
一見，hello-code.oの逆アセンブル結果と同じように見えますが，よく見るとアドレスの値が大きく変わっていることがわかります．_startのアドレスは，先ほどは0番地でしたが，今度は0x08048074です．msgの格納アドレスも0x08048098に変わり，これに合わせて3行目でECXレジスタに代入する即値も，同アドレスに変更されている点に着目しましょう．これは，Linuxのユーザープログラムが仮想記憶0x08048000番地にロードされる決まりになっているからです(_startが0x74番地後方から始まっている理由は，先頭にELFヘッダが存在するため)．

なお，0x8049098番地(msgの格納開始アドレス)からの逆アセンブル結果が無茶苦茶な内容になっていますが，これはobjdumpがデータ領域を誤って実行コードとして解釈しているためです．x86 CPUにとっても状況は同じであり，msgのエントリアドレスに対するジャンプ命令が指定されれば，x86はひたすら誤ったコードを実行し続けます．このように，メモリ上のバイトシーケンスはプログラマの腕一つで，実行コードにもデータにもなり得ることを肝に銘じておきましょう．

さて，fileコマンドがhello-code.oを解析した際に表示した“relocatable”という言葉を思い出してください．“relocatable”とは専門用語で「再配置可能」を意味しています．つまり，GNUアセンブラgasが出力するオブジェクトファイルは，最終的な配置アドレスは決定されておらず，「宙ぶらりん」の状態にあるのです．自由に配置可能な状態にある複数のオブジェクトファイルを，一定の順番で並べ直し，最終的なアドレ

〔図9〕helloasmの逆アセンブル

```
$ objdump -D helloasm

helloasm:     file format elf32-i386

Disassembly of section .text:

08048074 <_start>:
 8048074:       b8 04 00 00 00          mov    $0x4,%eax
 8048079:       bb 01 00 00 00          mov    $0x1,%ebx
 804807e:       b9 98 90 04 08          mov    $0x8049098,%ecx
 8048083:       ba 0f 00 00 00          mov    $0xf,%edx
 8048088:       cd 80                   int    $0x80
 804808a:       b8 01 00 00 00          mov    $0x1,%eax
 804808f:       bb 7b 00 00 00          mov    $0x7b,%ebx
 8048094:       cd 80                   int    $0x80
 8048096:       90                      nop
 8048097:       90                      nop
Disassembly of section .data:

08049098 <msg>:
 8049098:       48                      dec    %eax
 8049099:       65                      gs
 804909a:       6c                      insb   (%dx),%es:(%edi)
 804909b:       6c                      insb   (%dx),%es:(%edi)
 804909c:       6f                      outsl  %ds:(%esi),(%dx)
 804909d:       2c 20                   sub    $0x20,%al
 804909f:       77 6f                   ja     8049110 <__bss_start+0x68>
 80490a1:       72 6c                   jb     804910f <__bss_start+0x67>
 80490a3:       64 21 07                and    %eax,%fs:(%edi)
 80490a6:       0a 00                   or     (%eax),%al
```

〔図10〕リンクによるセクションの結合とアドレス配置の決定

スを決定するための作業がリンクです．リンクが終了して初めて，ファイルは“executable”となります(**図10**)．

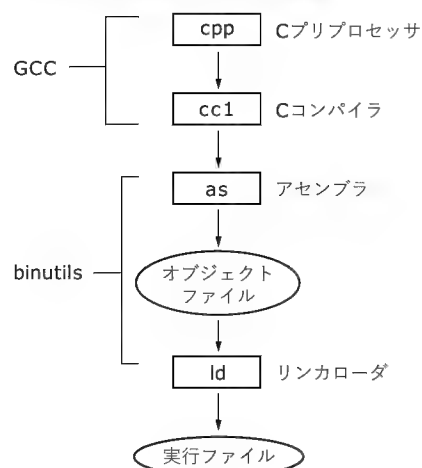
はじめにbinutilsありき

以上説明してきたとおり，実行コード生成のために最低限必要な役者は，「アセンブラとリンカ」この二つです．そして，as，ldおよび先程から紹介してきた各種のツール類はbinutils(BINary UTILitieS)パッケージに含まれています．GNU開発ツールと聞くと，Cコンパイラ(gcc)を思い浮かべる方が多いのではないかと思いますが，gcc(正確にはcc1)はGCC(Gnu Compiler Collection)パッケージ中に含まれる一コンパイラにすぎません．GNU開発環境を支えるもっとも重要な役者はbinutilsなのです(**図11**)．この点をしっかりおさえておきましょう．

Cコンパイラ登場

実行コードの中身と，その生成過程が理解できたところで，いよいよCコンパイラを使ったコード出力を行ってみます．おなじみの，hello.c(**リスト4**)を用意してください．

ソースの中身は説明するまでもありませんが，printf関数を使ってメッセージを表示する単純なコードです．1行目は，printf関数のプロトタイプ宣言を含んだstdio.hヘッダファイルをインクルードするためのものです．この行がなくてもエラーにはなりませんが，ソース中で未定義の関数を利用する場

〔図11〕GNU 開発ツールの全体像

〔リスト4〕hello.c

```
1  #include <stdio.h>
2
3  int main() {
4    printf("Hello, world!¥007¥n");
5    return 123;
6  }
```

〔図12〕ビルド

```
$ gcc -o hello hello.c
$ wc -c hello
  4144 hello
$ ./hello ;echo $?
Hello, world!
123
```

〔図14〕hello.iの内容(一部)

```
extern int printf (__const char *__restrict __format, ...) ;
...
# 2 "hello.c" 2

int main() {
  printf("Hello, world!¥007¥n");
  return 123;
 }
```

合には，必ずプロトタイプ宣言を用意し，単純なバグを防がなければなりません．それでは，さっそくビルドです(**図12**)．

ソースファイルhello.cを読み込み，実行ファイルを作成します．このとき，-o(Output)オプションにより，実行ファイル名はhelloに設定されます．ビルドが完了すると，4144バイト長のhelloが作成され，意図どおりに動作していることが確認できました．ここまではごく普通の光景ですが，じつはこの一見単純な作業の裏側にたいせつな工程が隠されているのです．

10 gccの正体

幸いgccコマンドの-v(Verbose)オプションを使うことで，ビルドの裏側で実行されている処理内容を詳細に追うことができます．チェックしてみましょう(**図13**)．

-save-tempsは，ビルド途中の一時ファイルを保存するためのオプションです．このオプションが指定されない場合，一時ファイルはビルド完了後に自動的に削除されます．

gcc -o hello hello.cを実行すると，まず最初にCプリプロセッサcppが起動され，hello.cソースファイルをhello.iに変換します．これを「プリプロセス処理」と呼びます．hello.i中には，stdio.hから芋づる式にインクルードされたヘッダファイルの膨大な内容が展開されています．hello.iのサイズは17Kバイトを超えますが，lessコマンドで実際にその内容を確認してみてください(**図14**)．

肝心のソースリストは最終尾に位置していますが，途中で

〔図13〕ビルドの裏側で実行されている処理内容

```
$ gcc -o hello hello.c -save-temps -v
Reading specs from /usr/lib/gcc-lib/i386-linux/2.95.4/specs
gcc version 2.95.4 20011002 (Debian prerelease)
 /usr/lib/gcc-lib/i386-linux/2.95.4/cpp0 -lang-c -v -D__GNUC__=2 -D__GNUC_MINOR__=95 -D__ELF__ -Dunix -D__i386__
   -Dlinux -D__ELF__ -D__unix__ -D__i386__ -D__linux__ -D__unix -D__linux -Asystem(posix) -Acpu(i386) -Amachine(i386)
                                                                    -Di386 -D__i386 -D__i386__ hello.c hello.i
GNU CPP version 2.95.4 20011002 (Debian prerelease) (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/lib/gcc-lib/i386-linux/2.95.4/include
 /usr/include
End of search list.
The following default directories have been omitted from the search path:
 /usr/lib/gcc-lib/i386-linux/2.95.4/../../../../include/g++-3
 /usr/lib/gcc-lib/i386-linux/2.95.4/../../../../i386-linux/include
End of omitted list.
 /usr/lib/gcc-lib/i386-linux/2.95.4/cc1 hello.i -quiet -dumpbase hello.c -version -o hello.s
GNU C version 2.95.4 20011002 (Debian prerelease) (i386-linux) compiled
                                                  by GNU C version 2.95.4 20011002 (Debian prerelease).
 as -V -Qy -o hello.o hello.s
GNU assembler version 2.13.2.1 (i686-pc-linux-gnu) using BFD version 2.13.2.1
 /usr/lib/gcc-lib/i386-linux/2.95.4/collect2 -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o hello /usr/lib/crt1.o
                          /usr/lib/crti.o /usr/lib/gcc-lib/i386-linux/2.95.4/crtbegin.o -L/usr/lib/gcc-lib/i386-linux/
                            2.95.4 hello.o -lgcc -lc -lgcc /usr/lib/gcc-lib/i386-linux/2.95.4/crtend.o /usr/lib/crtn.o
```

printf関数のプロトタイプが含まれている点に注目してください(lessの / コマンドで検索すると便利).

プリプロセス処理が終了すると，gccは次にCコンパイラcc1を起動します．cc1はhello.iを読み込んだ上で，これを「コンパイル」し，アセンブリソースhello.sを出力しますが，その内容は**リスト5**のとおりです．

hello-code.sとは若干様相が異なっていますが，基本骨格は変わりません．実行コードの前に，.rodataセクションを選択した上でメッセージデータを展開しています．Cコンパイラはメッセージデータのエントリアドレスに対して，自動的に.LC0というラベルを定義しています．

次に.textセクションを選択し，main関数のソースを展開しています．中央部でcall printf命令により，printf関数を呼び出していますが，直前のpushl $.LC0命令でメッセージデータの先頭アドレスをprintf関数の引き数として設定しています．これが「コンパイル」です．

gccはさらにアセンブラasを呼び出し，アセンブリソースhello.sをオブジェクトファイルhello.oに変換します．この過程が「アセンブル」です．ここで，hello.o内のシンボルをチェックしてみましょう(**図15**).

mainが登録されていることは当然ですが，printfが未定義になっている点に注目してください．これは，printf関数の実体がhello.o以外のファイル中に存在することを意味しています．その実体は一体どこにあるのか？　この疑問を忘れずに覚えておいてください．

もう一点，腑に落ちない点があります．教科書的には，「Cプログラムはmainから始まる」とされています．しかし本当にそうでしょうか．hello-code.sとhello.sの終了部分を注意深く比較してみてください．hello-code.s中では，プログラム終了前にexitシステムコールを呼び出していました．ところが，hello.s中のmain関数は仕事が終わると，機械語のret命令で呼び出し元にリターンするだけです．exitシステムコールは実行していません．ということは，main関数の呼び出し元がどこかに隠れていることになります．これが第二の疑問点です．

ビルド過程に戻ります．gccはアセンブルが終わると，最終的にcollect2(その実体はldコマンド)を呼び出します．collect2のコマンド行はもっとも引き数が多く難解ですが，整理するとcrt1.o/crti.o/crtbegin.o/hello.o/crtend.o/crtn.o計六つのオブジェクトファイル，および-lgcc，-lcオプションによりlibgcc(GCC専用ライブラリ)とlibc(glibc標準Cライブラリ)をリンクすることで，実行ファイルhelloを出力します．これが「リンク」です．gccによるビルドの全体像を**図16**に示します．

以上より，gccによるビルドの過程はプリプロセス/コンパイル/アセンブル/リンクの4工程を経ることが明らかになりました．gccはcpp/cc1/as/collect2計四つのコマンドの「呼び出し屋」にすぎない点を理解しておきましょう．

gccコマンドラインからのビルド制御

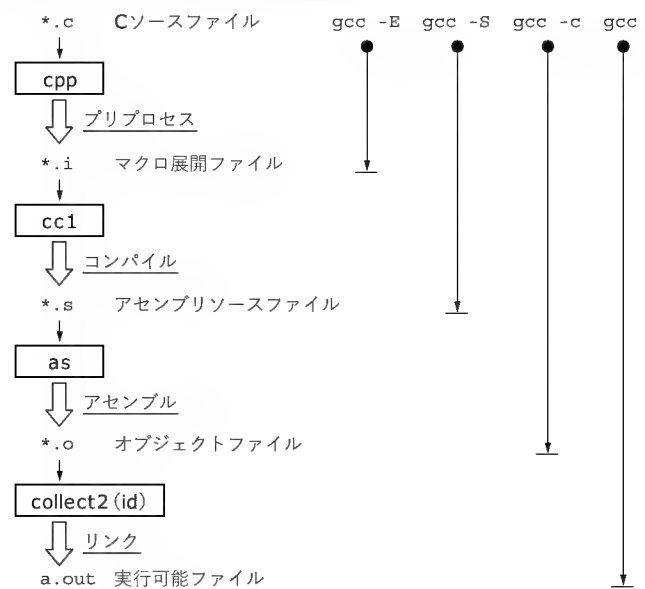
上で見たように，gccはデフォルトで実行ファイル生成まで行います．つまり，プリプロセス/コンパイル/アセンブル/リ

〔リスト5〕hello.s

```
        .file   "hello.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "Hello, world!¥007¥n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        addl $-12,%esp
        pushl $.LC0
        call printf
        addl $16,%esp
        movl $123,%eax
        jmp .L2
        .p2align 4,,7
.L2:
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.4 20011002 (Debian prerelease)"
```

〔図15〕hello.o内のシンボルをチェック

```
$ nm -g hello.o
00000000 T main
         U printf
```

〔図16〕gccによるビルドの全体像

〔表2〕ビルド工程を制御するための三つのオプション

-E	プリプロセス処理(cpp)のみを行う． 結果は標準出力に出力されるため，結果を保存したい場合はリダイレクションを用いる
-S	コンパイル処理まで(cpp/cc1)を行う． アセンブリソースは.sファイルに出力される
-c	アセンブル処理まで(cpp/cc1/as)を行う． アセンブル結果はオブジェクトファイル .o に出力される

ンクの全工程を一気に行ってしまうわけです．しかし，場合によってはオブジェクトファイルだけが欲しい場合もあるでしょうし，アセンブリソースをチェックしたいときもあるでしょう．このような目的のために，gccにはビルド工程を制御するための，三つのオプションが用意されています(**表2**)．

12 C Runtime Startup files

ここで，collect2コマンド行に現れたcrtで始まるオブジェクトファイル(crt1.o/crti.o/crtbegin.o/crtend.o/crtn.o)に注目してください．「crt」はC RunTime startupの略であり，main関数を呼び出す前後で特別な処理を担当しています．とくに重要なものは，先頭に位置しているcrt1.oであり，mainはこの中から呼び出されます．nmコマンドでcrt1.oの中身を見てみましょう(**図17**)．

crt1.o中にはELF実行ファイルのデフォルトエントリアドレスである_startが登録されています．さらに，mainシンボルが未定義になっている点に着目してください．これは，crt1.oの内部で外部シンボルであるmainを参照していることを示しています．つまり，crt1.oがmainの呼び出し元だったわけです．これで，一つ謎が解明しました．crtファイルには，C言語プログラム起動のための初期化処理，C++の場合はコンストラクタやデストラクタが格納されます．gccでビルドしたhelloのサイズが4144バイトもある背景には，リンクされたcrtファイル群が影響しているのです．

ふだんはcrtファイルの存在を意識する必要はありませんが，組み込みシステムなどでスタンドアロン式のプログラムを作成する場合は，自前のcrtファイルを用意しなければなりません．

13 ライブラリ

collect2の-lgcc，-lcオプションは，指定されたオブジェクトファイル中に関数や変数が存在しない場合，libgcc，libcライブラリを参照するよう指示しています．libgccはGCCに固有のライブラリであり，さまざまなグローバル変数や演算処理関数などが用意されています．libgccの本体は，少々わかりづらい場所に隠されていますが，gccコマンドに-print-libgcc-file-nameオプションを与えてみてください(**図18**)．

〔図17〕crt1.oの中身

```
$ nm /usr/lib/crt1.o
00000004 R _IO_stdin_used
00000000 D __data_start
         U __libc_start_main
         U _fini
00000000 R _fp_hw
         U _init
00000000 T _start
00000000 W data_start
         U main
```

〔図18〕libgcc.aの絶対パス

```
$ gcc -print-libgcc-file-name
/usr/lib/gcc-lib/i386-linux/2.95.4/libgcc.a
$ nm /usr/lib/gcc-lib/i386-linux/2.95.4/libgcc.a
```

〔図19〕/usr/lib/libc.a

```
$ nm /usr/lib/libc.a
...
printf.o:
00000000 T _IO_printf
00000000 T printf
         U stdout
         U vfprintf
...
```

〔図20〕lddコマンドの実行

```
$ ldd hello
        libc.so.6 => /lib/libc.so.6 (0x40017000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2
(0x40000000)
```

libgcc.aの絶対パスが表示されます．nmコマンドで，内部に登録されたシンボルを確認してみるとよいでしょう．次に，標準Cライブラリ(静的ライブラリ版)の実体は，/usr/lib/libc.aに相当します(**図19**)．

nmコマンドで内部シンボルを検索すると，printf.oオブジェクトファイル中に目的のprintf関数が含まれていることがわかります(ライブラリの実体であるアーカイブファイル.aは，複数のオブジェクトファイルを連結したもの)．これで二つ目の疑問が解消しました．printf.oがstdout変数とvfprintf関数を外部参照している点をチェックしておきましょう．

14 共有ライブラリ

以上で，UNIXにおける実行ファイルの内部構造とビルド方法の全体像は理解していただけたことと思います．しかし，実際にはもう一点理解しておかなければならない概念があります．これが共有ライブラリです．次のコマンドを実行してみてください(**図20**)．

lddは実行ファイルが「依存」しているライブラリやローダの一覧を表示するためのユーティリティです．libc.so.6は標

〔図21〕プログラムヘッダ情報の出力

```
$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x8048300
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x00478 0x00478 R E 0x1000
  LOAD           0x000478 0x08049478 0x08049478 0x0010c 0x00124 RW  0x1000
  DYNAMIC        0x00048c 0x0804948c 0x0804948c 0x000c8 0x000c8 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text
                                                                                                      .fini .rodata
   03     .data .eh_frame .dynamic .ctors .dtors .got .bss
   04     .dynamic
   05     .note.ABI-tag
```

準Cライブラリの共有ライブラリ版であり，ld-linux.so.2はELFプログラムローダです．実行ファイルがプログラムローダに依存しているというのは，おかしな話ですが，ELFではまず最初にプログラムローダが実行ファイルをメモリ上にロードし，外部依存(未定義)シンボルのアドレス解決を行ったうえで，プロセスを起動します．このように，実行ファイルがいったんプログラムローダで処理されることから，プログラムローダの別名は「インタプリタ」と呼ばれます．昔懐かしのBasicインタプリタと同じです．

ちょっと信じられませんが，readelfコマンドで-lオプションを指定し，プログラムヘッダと呼ばれる情報を出力してみてください(**図21**)．

プログラムヘッダは文字どおり，ELFファイルをプログラムとして実行するために必要な情報を記述したものですが，この中のINTERPヘッダ中に「プログラムインタプリタとして/lib/ld-linux.so.2を使用する」という記載があります．

すなわち，共有ライブラリに依存している実行ファイルは，対象となるライブラリはもちろん，プログラムローダも必要とするわけです．このため，libc.so.6やld-linux.so.2がいったん障害を受けると，これらに依存した実行ファイルはすべて起動不可能となってしまいます．これは，システム保守時に，共有ライブラリやプログラムローダのアップデートに失敗すると，二度とシステムを再起動できないことを意味しています．このため，安全性を重んじるUNIXでは重要なコマンド群を静的リンク版(後述)として作成し，/sbin/ディレクトリ内に一般アプリケーションとは明確に区別して保管しています．残念ながら多くのLinuxディストリビューションでは/sbin/ディレクトリのコマンドも共有ライブラリ版になっているので，ライブラリアップデート時には細心の注意をして作業するようにしましょう．ちなみにlibc.so.6の実体のサイズは1Mバイトを超えています．

静的リンク

最後に静的リンクについて，説明しておきましょう．静的リンクの反対語は動的リンクですが，共有ライブラリは動的リンクに基づいています．最初に紹介したhelloasmは静的リンクで生成されています．次のコマンドを実行してみてください(**図22**)．

先ほどのlddコマンドをhelloasmに適用すると，「これは動的な実行ファイルではない」と表示されました．言い換えれば「helloasmは動的ライブラリに依存していない」ことになります．この事実は，プログラムローダも必要しないことを意味していますが，readelf -lで確認すると，たしかにINTERPヘッダは見当たりません．

ダメ押しでfileコマンドを適用すると，今度は「helloasmは静的にリンクされたファイルである」と示されました．静的リンクとは，わかりやすく説明すると「何者にも依存せず，単独で実行可能なリンク形式」となります．安全性を優先するUNIX環境で，/sbin/ディレクトリ中のプログラムが動的リンクではなく，静的リンクを採用しているわけはここにあります．

「それでは，最初からすべてのコマンドを静的リンクでビルドすればよいではないか」と言われる方もいらっしゃることでしょう．たしかにそのとおりなのですが，危険を冒してまで共有ライブラリにこだわることにはわけがあります．ここで，gccの-staticオプションを使って静的リンクでhello.cから実行ファイルをビルドしてみましょう(**図23**)．

-staticは文字どおり，静的リンクを用いて実行ファイル

〔図22〕ldd コマンドを helloasm に適用する

```
$ ldd helloasm
        not a dynamic executable
$ readelf -l helloasm

Elf file type is EXEC (Executable file)
Entry point 0x8048074
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x00098 0x00098 R E 0x1000
  LOAD           0x000098 0x08049098 0x08049098 0x00010 0x00010 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data
$ file helloasm
helloasm: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
```

〔図23〕静的リンクで hello.c から実行ファイルをビルドする

```
$ gcc -static -o hello hello.c
$ file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
$ ldd hello
        not a dynamic executable
$ wc -c hello
 461436 hello
```

をビルドするよう gcc に指示するためのオプションです．file コマンドにより，今回の hello は「静的にリンク」されていることがわかります．ldd コマンドの結果も，依存しているライブラリはなく，hello が動的にリンクされていない事実を示しています．

最後に，静的リンクで生成された hello のファイルサイズをチェックしてみてください．なんと 450K バイトという，動的リンク版の 100 倍にもおよぶ大きさです．先ほど，printf.o が stdout 変数や vfprintf 関数を外部参照していると書きましたが，たった一つの printf 関数をリンクするだけで，ここまで膨大な「負債」を被ることになるのです．しかも，一つの実行ファイルだけでなく，200 のシステムツールが静的リンクで作成されたとすると，その合計サイズは軽く 90M バイトを超えてしまいます．

これではあまりにも無駄が多すぎるため，頻用されるライブラリ関数などは共有ライブラリを通じて，可能な限り共用しようという考えが生まれました．共有ライブラリは仮想記憶上の一箇所にロードされると，複数のプロセスから参照可能となるよう設計されています．この結果，100 種類のプロセスが printf 関数を呼び出しても，その実体はただ一箇所に存在するだけでよいのです．

おわりに

以上，駆け足でしたが GNU 開発ツールを使いこなすために必要な基礎知識を紹介しました．binutils や GCC パッケージ中には，このほかにも数え切れないほどの機能が盛り込まれています．ぜひ，一つ一つのツールを手に取りながら，ビルドの過程を自分の目で追ってみてください．最初は回り道に思えるかもしれませんが，GNU 開発ツールは将来必ずや，皆さんの右腕として活躍してくれることと思います．

にしだ・わたる

オープンソースのITRON仕様OS

TOPPERSで学ぶRTOS技術

第1回 TOPPERSプロジェクトの概要と展開

高田広章

はじめに

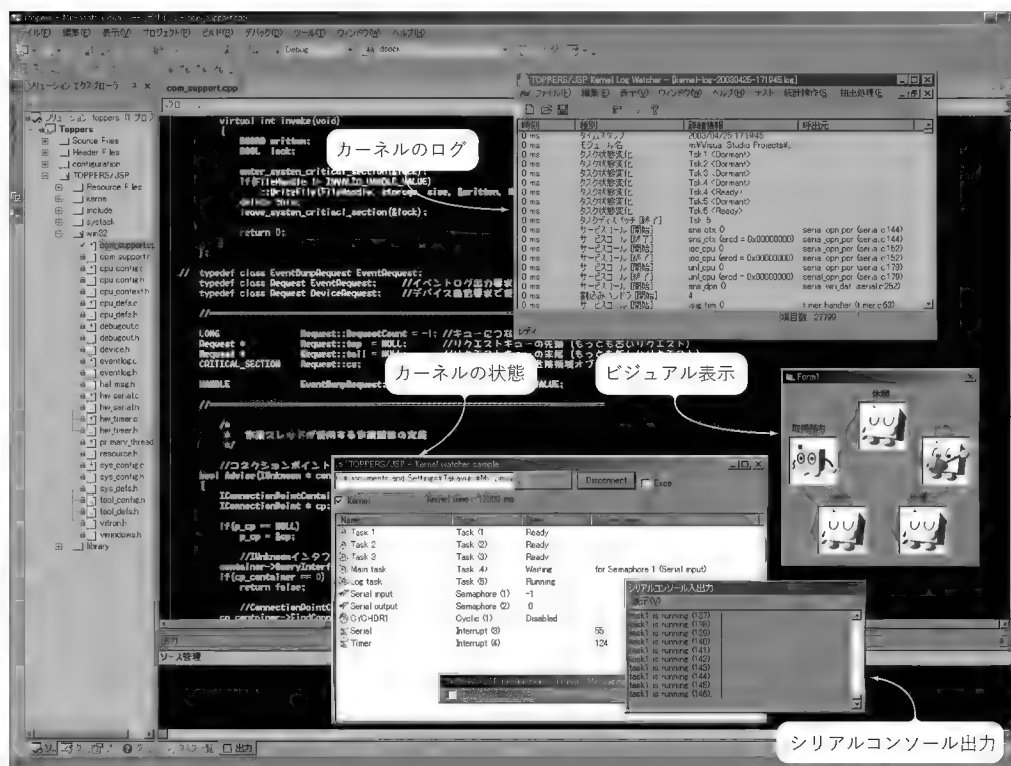
この号から「TOPPERSで学ぶRTOS技術」と題する連載を開始することになりました．この連載は，μITRON4.0仕様に準拠したオープンソースのリアルタイムOS（RTOS）であるTOPPERS/JSPカーネルなど，TOPPERSプロジェクトの開発成果を題材に，RTOSとその周辺技術を解説していこうというものです．

TOPPERS/JSPカーネルのソースコードは，TOPPERSプロジェクトのWebサイト（http://www.toppers.jp/）からダウンロードでき，シミュレーション環境はパソコン上で動作するので（**写真1**），この連載をきっかけに，ぜひご自分で動かしてみていただきたいと思います．

次回以降は，TOPPERSプロジェクトのメンバが持ち回りで執筆を担当し，プロジェクトの開発成果の活用方法の解説や，カーネルのポーティング過程の紹介などを通して，教科書的にも使える連載にしたいと考えています．また，ITRON仕様に関連する標準化動向の解説や，ITRON仕様を利用する上でのノウハウや仕様策定の理由などについても，コラム的に紹介していきたいと考えています．

第1回の今回は，TOPPERSプロジェクトのねらいや目的，これまでの成果や今後の計画と，TOPPERSプロジェクトの最初の開発成果であるTOPPERS/JSPカーネルについて紹介します．また現在，TOPPERSプロジェクトをNPO法人として組織化する準備を進めており，この6月から会員募集を開始したので，そのことについても紹介します．

〔写真1〕Windows上のシミュレーション環境の動作例

この画面イメージは，JSPカーネルのWindows上のシミュレーション環境の動作例．背景にあるのがVisual C++のウィンドウで，手前には，カーネルの現在状態を表示するウィンドウ，カーネルのログを表示するウィンドウ，シリアルI/Oに送信した文字を表示するためのウィンドウ，プログラムの動作状態をビジュアルに表示するためのウィンドウがある．最後のウィンドウを表示しているプログラムは，Visual Basicで記述してあり，JSPカーネルのシミュレータとはCOMを使って通信している．

TOPPERSプロジェクトとは？

TOPPERSプロジェクトは，組み込みシステム構築の基盤となる各種のソフトウェアを開発し，自由に利用できる良質なオープンソースソフトウェアとして公開すること，またその利用技術を提供することにより，組み込みシステム技術ならびに産業の振興を図ることを目的としたプロジェクトです．このプロジェクトは，筆者らの研究室（名古屋大学大学院 情報科学研究科 組込みリアルタイムシステム研究室）を中心に，プロジェクトの趣旨に賛同してソフトウェア開発を分担する組織や個人の協力を得て推進しています．

組み込みシステム構築の基盤となるソフトウェアとしては，まずRTOSが重要であり，TOPPERSプロジェクトにおいては，ITRON仕様のリアルタイムカーネルの開発を中心に進めています（**コラム1**参照）．多様な要求事項をもったさまざまな組み込みシステムに対応するために，各種のリアルタイムカーネルを開発するとともに，ソフトウェア部品（ミドルウェア）や開発環境の整備も進め，ゆくゆくは，組み込みシステム分野において，Linuxのような位置付けとなるOSに育てていきたいと考えています．

ちなみに“TOPPERS”は，“Toyohashi OPen Platform for Embedded Real-time Systems”の略で，「トッパーズ」と読みます．最初の“Toyohashi”はいうまでもなく地名ですが，プロジェクトの開始時に筆者が豊橋技術科学大学に所属していたことから付けた名前です[注1]．ちなみに英語的には，“toppers”は“topper”の複数形で，“topper”には「卓越するもの」，「すぐれたもの」といった意味があります．

ITRON仕様

TOPPERSプロジェクトは，ITRON仕様の成果を技術的な基盤としています．ITRON仕様については本誌でも繰り返し紹介されているので，ここでは簡単な紹介に留めたいと思います．

ITRON仕様は，トロンプロジェクトにおいて標準化を進めてきたリアルタイムカーネル仕様です．ITRON仕様の標準化は約20年前に始まり，現在までに4世代のリアルタイムカーネル仕様を策定・公表してきました（**図1**）．最新のバージョンであるμITRON4.0仕様は，現世代のリアルタイムカーネル技術の範囲では，きわめて完成度の高い仕様に仕上がっていると自負しています．

ITRON仕様は，「仕様」とあることからもわかるとおり，あくまでもRTOSの機能やAPIを規定しているものであり，ITRONというRTOSがあるわけではありません．ITRON仕様

コラム1 リアルタイムOS（RTOS）とリアルタイムカーネル

リアルタイムカーネルとは，カーネル（核）という言葉が示すとおり，もともとはRTOSの中心となるモジュールのことをいいます．具体的には，プロセッサ，メモリ，タイマなど，どのようなシステムにも共通する資源を扱うシステムソフトウェアを，リアルタイムカーネルと呼びます．逆にいうとリアルタイムカーネルは，ファイルシステムや各種プロトコルスタックなど，I/Oデバイスを扱うための機能はもっていません．

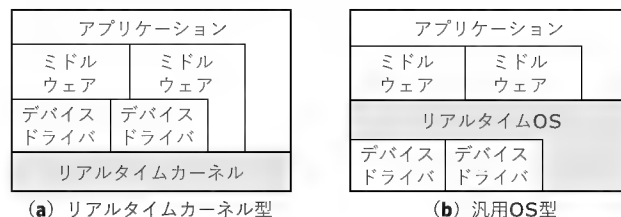
ITRON仕様などの小規模な組み込みシステム向けのRTOSは，リアルタイムカーネル部分のみで構成されているのが一般的です．これは，それぞれの組み込みシステムがもつI/Oデバイスが異なっており，RTOSが特定のI/Oデバイスをサポートする意義が低かったためです．たとえば，RTOSがファイルシステムをサポートしても，それを必要としない組み込みシステムも数多くあります．このようなRTOSでは，I/Oデバイスを扱うソフトウェアは，リアルタイムカーネル上で実現することになります〔**図A**（**a**）〕．このようなタイプのRTOSを，筆者はリアルタイムカーネル型と呼んでいます．

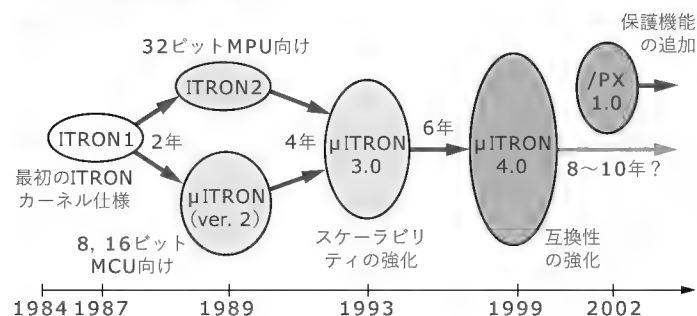
それに対して，組み込み向けのLinuxやWindows CEなどは，汎用システム向けのOSの構造そのままで，RTOSを実現しています〔**図A**（**b**）〕．そのため，I/Oデバイスをあつかうソフトウェア（デバイスドライバやファイルシステム，プロトコルスタックなど）は，RTOSの中に抱えている構造をしています．

リアルタイムカーネル型のRTOSでは，RTOSと称していてもじつはリアルタイムカーネルにほかならず，この場合には，二つの言葉は同じ意味で使われます．本文でも，RTOSとリアルタイムカーネルという言葉を明確な区別をせずに用いていますが，あえていえば，リアルタイムカーネル部分のみのRTOSのことを，誤解がないようにリアルタイムカーネルと呼んでいると考えてください．

〔図A〕汎用OS型のリアルタイムOSとリアルタイムカーネル

（**a**）リアルタイムカーネル型

（**b**）汎用OS型

注1：筆者が名古屋大学に移ったことから，TOPPERSという名前を変えないのかという質問をよく受けるが，知名度も徐々に上がってきているので，名前を変えるつもりはない．

〔図1〕ITRON仕様の歴史

に準拠したRTOSは数多く開発されています．ITRON仕様は，誰もが自由にそれに準拠したRTOSを開発・販売できるという意味で「オープンな仕様」ですが，それに準拠して開発されたRTOSはオープンであるとは限りません．本誌の広告を見ていただいてもわかるとおり，実際に多くのITRON仕様準拠のRTOSが販売されています．

ITRON仕様のリアルタイムカーネルは，とくに小規模な組み込みシステムを中心に広く用いられており，ITRON仕様の標準化団体であるトロン協会のアンケート調査によると，国内で開発される組み込みシステムの30〜40%に用いられているという結果が得られています．

またトロンプロジェクトでは，リアルタイムカーネル仕様に加えて，各種のソフトウェア部品の仕様（JTRON仕様やITRON TCP/IP API仕様など）やソフトウェア開発環境に関連する仕様（ITRONデバッギングインタフェース仕様）の標準化も行っています．これらの仕様も加えて，ITRON仕様と総称することがあります．

組み込みシステムの大規模化とITRON仕様の問題点

ITRON仕様は，仕様の名称についている「μ（マイクロ）」という文字が示しているとおり，小規模な組み込みシステムに向いたコンパクトでオーバヘッドの小さいリアルタイムカーネル仕様です．実際，コンシューマ製品を中心とする小規模な組み込みシステムの分野で広く用いられています．

ところが近年，半導体技術の急速な進歩が後押しする形で，組み込みシステムの複雑化が進んでおり，携帯電話のような「小さい」組み込みシステムにも，非常に「大規模な」ソフトウェアが組み込まれるようになってきました．そのため，ITRON仕様を大規模な組み込みソフトウェア開発にも適するように発展させることが求められています．

具体的な課題として，小規模な組み込みソフトウェア開発においてはカーネル機能のみあれば十分であったのに対して，大規模なソフトウェア開発では，プロトコルスタックやファイルシステム，各種のデバイスドライバなどのソフトウェア部品がそろっていることが要求されます．また，優れたソフトウェア開発環境が提供されていることも重要です．

前述のトロン協会による調査でも，ITRON仕様の問題点として「開発環境やツールが不足」や「ソフトウェア部品が不足」を挙げる技術者が多くいます．また，「扱える技術者が少ない」，「ソフトウェアの移植性が悪い」といった問題も数多く指摘されています．しかし，RTOS自身の技術的な問題を指摘する声は多くありません．

われわれは，これらの問題のおもな原因が，過剰な重複投資と過剰な多様性にあると考えています．現在ITRON仕様OSを開発している企業は10社以上ありますが，それらの企業のほとんどが，その上で動作するソフトウェア部品を開発しています．そのため，ソフトウェア部品に対する開発投資が少ないわけでも，開発技術者が少ないわけでもありません．それにも関わらずソフトウェア部品が不足しているという指摘があるのは，各社がTCP/IPプロトコルスタックなど需要の高いソフトウェア部品を独立に開発しており，その結果，TCP/IPプロトコルスタックは10種類あるが，先端的なソフトウェア部品になると，どの企業も開発していないという大きな落差を生じています．これが，過剰な重複投資の問題です．

開発環境やツールの不足についても，類似のことがいえます．また，各社の開発するITRON仕様OSの機能や内部構造はそれぞれ異なるため，ツールが対応したとしても，特定のITRON仕様OS専用になってしまうおそれがあります．このことが，ツールがなかなかITRON仕様OSに対応しないという事態につながります．これが，過剰な多様性の問題です．

TOPPERSプロジェクトのねらい

筆者がTOPPERSプロジェクトを進めている大きな動機は，日本の主要産業分野で重要な役割を果たしている組み込みシステムの分野で，日本独自のITRON仕様の技術を維持・発展させていきたいということです．

そのためには，上で述べた過剰な重複投資と過剰な多様性の問題を解決/軽減しなくてはなりません．われわれは，ITRON仕様OSの標準的な実装をオープンソースソフトウェア化することにより，少なくともカーネルに対する過剰な重複投資の問題は軽減されると考えています．また，ITRON仕様OSの実装の種類が減ることになれば，ITRON仕様OS上で動作するソフトウェアの移植性が向上し，また，ソフトウェア部品や開発環境の対応がそれらに集中する効果があります．

これに加えてTOPPERSプロジェクトでは，ITRON仕様をベースとして，次世代のRTOS技術を確立することもねらっています．オープンソースソフトウェア化により，産学官の力を結集することが容易になります．また，これまでITRON仕様の策定で採ってきたアプローチ（まず標準仕様を策定し，その後に各社でソフトウェアを開発）より，直接ソフトウェアを開発したほうが，技術開発のスピードが速いと予想されます．

TOPPERSプロジェクトで開発するRTOS技術は，「ITRONの良さ」を継承したものであること，言い換えると，組み込みシステムの要求に合致した技術でなければならないと考えています．先に「Linuxのような位置付けとなるOSに育てていきたい」と書きましたが，Linuxをもう一つ作るつもりはないということです（Linuxは，汎用システム向けのOSとして開発されたものである）．

さらにわれわれは，組み込みソフトウェア技術者の育成が重要な課題になっていると認識しています．オープンソースソフトウェアは教材としては適したものであり，TOPPERSプロジェクトでも，教材の提供や教育の場を設けるなどの活動を通じて，技術者教育に貢献していきたいと考えています．

以上からおわかりと思いますが，TOPPERSプロジェクトでは，「オープンソース」を組み込みシステム技術および業界を発展させるための手段と考えており，オープンソースそれ自身が目的ではないことをご理解ください．

TOPPERSプロジェクトの経緯とNPO法人化

TOPPERSプロジェクトは，次に紹介するTOPPERS/JSPカーネル（以下，JSPカーネルと略す）の最初のバージョンを配布開始した時点（2000年11月）から始まります．この時点では，上で述べた過剰な重複投資と多様性の問題の解決/軽減といったことは意識しておらず，大学での開発成果を産業界で活用してもらいたいという軽い気持ちでした．

その後，いくつかの組織から，JSPカーネルやその上で動作するソフトウェア部品の開発に一緒に取り組みたいという申し出があり，配布開始から1年後の2001年11月には，4組織からなるプロジェクトとなりました．また，ITRONの関係者と議論する中で，JSPカーネルが産業界で受け入れられる可能性が十分にあること，ITRON仕様OSの標準的な実装をオープンソース化することがITRON仕様の抱える問題の解決・軽減につながることを意識しはじめ，産業界へのプロモーションを本格化させました．その結果，翌2002年4月には，組み込みシステム業界の有力4社に参加いただき，さらにその後も着実に参加メンバを増やし，現時点での参加メンバ数は20弱となっています．

また，この間の2001年5月には，JSPカーネルの開発成果が評価され，第3回LSI IPデザイン・アワードにおいてIP優秀賞を受賞しました．受賞者は，最初のバージョンの開発に参加した3名（高田広章，若林隆行，本田晋也）です．

これまでTOPPERSプロジェクトは，大学の研究室を実質的な事務局とする小さなボランティア組織で運営してきたわけですが，ソフトウェアの開発・普及を促進するためには，予算をもった組織が必要となってきました．また，参加メンバが増えるに連れて，事務局なしでの運営では支障をきたすようになってきました．

そこで，2002年11月にTOPPERSプロジェクト組織化準備委員会を設けて検討してきた結果，TOPPERSプロジェクトをNPO法人として組織化することとなり，6月には新組織（NPO法人）の会員募集を開始しました．

TOPPERSプロジェクトに何らかの形で関与したいという方は，ぜひ入会をご検討くださると幸いです．また，TOPPERSプロジェクトを応援したいというだけでも歓迎です．詳しいことは，TOPPERSプロジェクトのWebサイト（`http:// www.toppers.jp/`）をご参照ください．

TOPPERS/JSPカーネル

TOPPERSプロジェクトの最初の開発成果が，TOPPERS/JSPカーネルです．JSPカーネルは，μITRON4.0仕様のスタンダードプロファイル規定[注2]に準拠したリアルタイムカーネルです．“JSP”は“Just Standard Profile”の略なのですが，“Just”という語が示すとおり，スタンダードプロファイル規定「ちょうど」の機能をもつように設計したものです（実際には，若干の拡張機能がある）．

- ターゲットプロセッサ

JSPカーネルは，後でも紹介するとおり，新しいターゲットシステムへのポーティングが容易な構造となっており，数多くのプロセッサで動作します．具体的には，2002年4月に配布を開始したRelease 1.3ではMC68040，SH-3/SH-4，SH-1，H8，H8S，ARM7/ARM9，V850，M32R，MicroBlaze，TMS320C54x，i386をサポートしており，Windows上とLinux上で動作するシミュレーション環境を用意しています．また，ユーザー側で新しいプロセッサへポーティングした（ないしは，ポーティングしている）という報告もいくつかいただいています．さらに，Nios，MIPS，PowerPC，M16Cなどへのポーティングも進めており，今年度内には，主要な組み込みシステム向けプロセッサの大半をカバーできる予定です．

- 開発の目的

JSPカーネルの開発の目的は，多岐にわたります（**表1**）．まず挙げなければならない一義的な目的は，大学や高専などの研究・教育機関における研究・教育のプラットホームとしての利用です．筆者の研究室では，JSPカーネルを研究・教育にフル活用しているのはいうまでもありません．

次に，μITRON4.0仕様の策定責任者だった筆者にとっては，

注2：μITRON仕様では，最低限の要件さえ満たしていれば，仕様に定義されている機能のうち，どれだけの機能を実装するかは実装者にまかされている．つまり，μITRON仕様準拠のカーネルといっても，どれだけの機能をもっているかは実装ごとに異なる．ただし，これでは互換性を保つことが非常に難しくなるため，ある規模のシステムを想定して標準的に実装すべき機能セットを厳密に定義している．これをプロファイル規定と呼び，μITRON4.0仕様では，スタンダードプロファイルと自動車制御用プロファイルの二つのプロファイルを規定している．

μITRON4.0仕様の評価も重要な開発目的でした．とくに，μITRON3.0仕様から追加したタスク例外処理機能とオーバランハンドラ(**コラム2**)については，実際に実装を行って，どの程度のオーバヘッドになるかを評価する必要がありました．この評価結果については，次の論文で報告したので，興味をお持ちの方はぜひご参照ください．

〔表1〕TOPPERS/JSPカーネルの開発の目的

●研究・教育機関における研究・教育のプラットホーム
●μITRON4.0仕様の評価
●μITRON4.0仕様のリファレンス実装
●μITRON4.0仕様カーネル上のソフトウェア部品開発のプラットホーム
●評価目的・プロトタイプ開発・テスト環境への利用
●実製品への適用

〔写真2〕松下電器のカラオケマイク「DO!KARAOKE」

パナソニックSDカラオケマイク「SY-MK7-S」
デュエットマイク「SY-DK7-S」
(2003年2月　松下電器)

本田晋也，高田広章：「μITRON4.0仕様における例外処理機能とその評価」，『情報処理学会論文誌』，vol.42，no.6，pp.1514-1524(2001年6月)

また，産業界において活用していただくことも，当初から目的の一つとしています．すでにμITRON仕様の評価目的やプロトタイプ開発への利用，テスト環境への利用については，いくつかの企業で利用が始まっています．とくにWindows/Linux上でのシミュレーション環境は，ターゲットシステムに組み込む前にソフトウェアの論理レベルのテストを行うのにきわめて有効なものです．さらには，実製品への適用も進んでおり，製品化された機器に組み込まれた例としては，松下電器のカラオケマイク「DO!KARAOKE」があります(**写真2**)．

そのほかにも，μITRON4.0仕様のリファレンス実装や，μITRON4.0仕様カーネル上のソフトウェア部品開発のプラットホームとして利用いただくことも，開発の目的に挙げることができます．

ここで注意していただきたいのは，JSPカーネルの開発そのものを大学の研究活動の一環とは位置付けていないことです．研究活動としてOSを開発すると，新規性を強調するために，いびつな実装になってしまいがちです．われわれは，JSPカーネルそのものには新規性を求めず，新規なアイデアを盛り込むためのベース(ないしは，プラットホーム)と位置付けています．

● おもな特徴

JSPカーネルのおもな特徴は次のとおりです．

▶読みやすく改造しやすいソースコード

研究・教育への利用を一義的な目的として開発したことから，ソースコードの読みやすさや改造しやすさに重点を置いて実装しています．定量的な評価は難しいですが，ソースコードの読みやすさには自信をもっています．

ただし，安易な読みやすさのために，効率の悪い平易なアルゴリズムを使用することはしていません．むしろ，タイムイベ

コラム2 オーバランハンドラ

与えられた時間制約を厳密に守って動作しなければならないハードリアルタイムシステムでは，それぞれのタスクの最大実行時間を見積もり，さらにそれを積み上げてシステム全体が時間制約を守れるかを検証する手法が使われます．ここで，タスクの最大実行時間もの見積もりが厳密に行えればよいのですが，実際には厳密に見積もろうとすると実際の最大値よりもかなり悪い(つまり悲観的な)値になってしまいます．

そこで，時間制約がゆるやかなシステムでは，各タスクにもち時間を与え，その範囲内で動作させようという手法が考えられます．タスクがもち時間を越えて実行を続けようとしたら，そこで例外処理を行うわけです．タスクがもち時間を越えて実行することを，オーバランと呼びます(直訳すると，「走り過ぎ」)．

オーバランハンドラは，タスクが設定されたもち時間を越えて実行した場合に起動されるハンドラで，μITRON4.0仕様で新たに導入された機能です．タスクに対して，その上限プロセッサ時間を設定しておくと，カーネルはタスクが使用したプロセッサ時間を計測し，上限プロセッサ時間を越えると登録しておいたハンドラを起動します．

μITRON4.0仕様では，オーバランハンドラで例外処理を行うのは，アプリケーション側の責任としています．そこで，問題となったタスクを強制的に中断する手もありますし，そのタスクの優先度を下げて後回しにする手もあります．また，システム異常と判断してシステムをリセットする方法も考えられます．アプリケーション側の責任としているのは，どのような対処法が妥当であるかがアプリケーションごとに異なるためです．

ント管理にヒープ構造を使うなど，複雑であっても効率的なアルゴリズムは積極的に採用しています．

実際，JSPカーネルをベースにして，さまな拡張や改造が行われています．すでに述べたように，μITRON4.0仕様の評価を目的にオーバランハンドラ機能を実装したことに加えて，ミューテックス機能の実装やマルチプロセッサ対応などをすでに実験的に行いました(**写真3**)．また，サービスコールを遅延実行するように改造したという報告を，ユーザーの方からいただいています．

▶他のターゲットへのポーティングが容易な構造

他のターゲットプロセッサやシステムへのポーティングの容易さを重視した設計をしました．具体的には，ターゲット非依存部とターゲット依存部を明確に分離し，ターゲット非依存部に関してはすべて標準のC言語で，ターゲット依存部に関してもできる限り多くの部分をC言語で記述しています．

とくに考慮したのが割り込み処理です．割り込み処理は，実行時性能を向上させる上では非常に重要なポイントとなる一方で，プロセッサのもつ機能の違いが大きく，プロセッサの違いを安易に隠蔽すると実行時性能の低下につながります．JSPカーネルでは，実行時性能の低下を最低限に留めて，プロセッサのも持つ割り込み処理機能を抽象化しました．最初からJSPカーネルのソースコードを読むと，きわめて当たり前の方法に見えるかもしれませんが，ここに辿りつくまでにかなりの試行錯誤をしています．

また，他のターゲットへのポーティングを容易にするために，ターゲット依存部のインターフェース仕様も公開しています．その結果，最初にJSPカーネルを公開した1週間後には，i386(PC)へポーティングしたというユーザーがあらわれました．これには，JSPカーネルを開発したわれわれにも望外の驚きで，オープンソースの力を認識させられました．ポーティングを行ったユーザーから，ポーティングにかかった時間は実質3日間という報告を受けており，JSPカーネルが他のターゲットへのポーティングしやすい構造になっていることが証明されたと考えています(3日間という時間は，ターゲットプロセッサや開発環境のことをよく理解していることが前提．実際には，それらを理解するのにより長い時間がかかってしまう)．

▶高い実行性能と小さいRAM使用量

大部分がC言語で記述されているカーネルとしては，高い実行時性能と少ないRAM使用量を実現しました．さすがに，すべてアセンブラで記述されたカーネルには，実行時性能の面で勝てないと思います(どの程度の違いがあるのか，一度実際に測定したいと思っている)．

とくにRAM使用量の削減を重視した設計にしており，32ビットプロセッサ向けの典型的な実装では，タスクコントロールブロック(TCB)のサイズを32バイトに抑えています．それに対してROM使用量の削減はあまり重視しておらず，まだ最適化の余地が残っています．

〔写真3〕マルチプロセッサ対応カーネルの開発環境

この写真は，筆者の研究室で進めている，JSPカーネルをマルチプロセッサ対応に拡張するための開発環境．用いているボードは，Xilinx社のFPGAのまわりにSRAMなどを載せたもので，この研究用に開発してもらったもの．写真は，ボード上のFPGA上に四つのMicroBlazeプロセッサを載せ，それぞれでJSPカーネルを動作させているところ．ボードと開発環境は8本のシリアルケーブルで接続されているが，これは，各プロセッサごとに2本(デバッガ用とアプリケーションプログラムからのデータ出力用)のシリアルポートを用いているため．

▶Windows上およびLinux上でのシミュレーション環境

JSPカーネルには，Windows上およびLinux上で動作させるためのシミュレーション環境が含まれています．具体的には，Windows/Linuxの一つのプロセスの中で，複数のタスクを切り換えて実行することで，JSPカーネルの動作をシミュレートするものです．ここで強調しておきたいことは，これらのシミュレーション環境は，JSPカーネルのターゲット非依存部にいっさい手を加えず，ターゲット依存部のみを入れ換えることで実装していることです．そのため，ターゲットシステムの動きをかなり正確にシミュレートできます．

Windows上のシミュレーション環境は，ITRONのタスク一つに対してWindowsのスレッドを一つ用意し，実行状態のタスクに対応するスレッド以外はサスペンドさせる方法をとっています．そのため，Visual Studioなど，Windowsのマルチスレッドに対応した開発環境が，そのままITRONのマルチタスクに対応した開発環境として使えます．

上でも述べましたが，これらのシミュレーション環境は，ターゲットハードウェアが完成する前のプロトタイプ開発や論理レベルでのテストに有効なものです．また，ハードウェアの完成後でも，開発に使えるハードウェアの数は限られているのが通常でしょうから，開発者が多い場合には有効に活用できると考えています．さらに，パソコン1台で動作するので，RTOSの学習用途にも適しています．

▶開発環境まで含めてフリーソフトウェアのみで構築可能

JSPカーネルは，GNU開発環境を標準のソフトウェア開発環

境としています．そのためユーザーは，カーネル本体のみならず，開発環境もフリーで入手し，システム開発を行うことが可能です．

● 開発の経緯

JSPカーネルの開発は，μITRON4.0仕様の標準化が完了した1999年の半ばに開始し，その年末までには基本部分が完成しました．その後，シミュレーション環境やドキュメントの整備を行い，上述のように2000年11月に最初のバージョンの配布を開始しました．その後数回のバージョンアップを行い，現時点ではRelease 1.3が最新のバージョンとなっています．また，本誌が発行される頃にはRelease 1.4を出す予定です．

JSPカーネルの開発に着手する数年前から，筆者らは教育・研究用途をおもな目的としてμITRON3.0仕様に準拠したリアルタイムカーネルを開発しており，ItIs（ITRON Implementation by Sakamura Lab.）という名称で配布していました．μITRON4.0仕様に準拠したカーネルを開発するにあたり，ItIsをバージョンアップする方法も考えましたが，次のような理由から新たに開発しなおすことにしました．

▶ ItIsの開発時には，実製品への適用までは想定していなかったため，実行性能よりも，読みやすさや改造しやすさを重視して実装しました．JSPカーネルでは，機能をスタンダードプロファイルに絞り込むことで，実行性能と読みやすさ・改造しやすさの両立を目指すことにしました．

▶ ItIsには，ITRON仕様に対する各種の拡張機能を盛り込んだため，ソースコード中に条件コンパイル指定（#ifや#ifdef）が多数含まれることになり，RTOSの初心者にとっては非常に読みにくいものでした．この点についても，機能をスタンダードプロファイル「ちょうど」と決めることで，ほとんどの条件コンパイル指定を取り除くことができ，初心者にも読みやすいものとすることを目指しました．

▶ ItIsは当初，トロン仕様のマイクロプロセッサをターゲットとして開発し，トロン仕様プロセッサのもつリアルタイムカーネル向けの特殊な機能を活用した構造を採っていたため，他のプロセッサにポーティングした場合に効率が悪いものとなっていました．JSPカーネルでは，より一般的なプロセッサで効率が出るような構造を採用することとしました．

JSPカーネルからの発展形

JSPカーネルは，スタンダードプロファイルちょうどの機能と決めているので，引き続きバージョンアップしているといっても，サポートするターゲットシステムが増えたり，ポータビリティが上がるといった改良が中心で，機能拡張はしていません．

一方，上述したように組み込みソフトウェアの大規模化は急速に進んでおり，それに対応するための進んだ機能をもったRTOSは，JSPカーネルからの発展形として，別に開発しています．ここでは，JSPカーネルからの発展形としてこれまでに開発が完了している，IIMPカーネルとIDLカーネルについて紹介します．

● IIMPカーネル

最近，組み込みソフトウェアの大規模化にもかかわらず，開発期間短縮に対する要求も厳しいことから，組み込みソフトウェアの品質や信頼性の確保が大きな問題となっています．組み込みソフトウェアの品質・信頼性の問題を解決もしくは軽減するための一つのアプローチとして，RTOSに保護機能を導入する方法が考えられます．

OSの保護機能とは，OS上で動作するアプリケーションソフトウェアにバグがあった場合に，そのバグに起因する障害が，OSや他のアプリケーションに波及するのを防ぐための機能のことをいいます．メモリ保護機能は，各アプリケーションが他のアプリケーションのメモリ領域を破壊するのを防ぐための機能で，OSのもつ保護機能としてもっとも典型的なものです．

メモリ保護機能は，汎用システム向けのOSにおいては一般的ですが，小規模なリアルタイムOSのほとんどはもっていません．これは従来，(a)保護機能を実現するためのオーバヘッドが大きいこと，(b)ソフトウェアが小規模で高い信頼性を確保することが比較的容易であったこと，(c)機器の出荷後に外部からソフトウェアがダウンロードされることはなく，ソフトウェアのデバッグが終わるとシステム内で動作するソフトウェアはすべて信用できるという前提が成り立ったこと，という三つの理由によると考えられます．しかし，最近の組み込みシステムでは，この中の(b)と(c)の条件が成立しなくなってきています．

トロン協会では，このような背景から，μITRON4.0仕様をベースにメモリ保護機能を導入するための検討を行い，その結果，2002年6月にμITRON4.0仕様 保護機能拡張（略称：μITRON4.0/PX仕様）を公開しました．

この仕様検討と並行してトロン協会では，情報処理振興事業協会（IPA）による情報技術開発支援事業の採択テーマとして，検討中の仕様に準拠したリアルタイムカーネルの開発を行いました．これがIIMPカーネルで，JSPカーネルをベースに保護機能を拡張する形で実装しました．ちなみにIIMPは，“An Implementation of ITRON with Memory Protection”の略です．

IIMPカーネルの特徴は，μITRON4.0/PX仕様の特徴でもあるのですが，一言でいうと，組み込みシステムの要求に合致したオーバヘッドの小さいメモリ保護機能を実現したことです．具体的には，組み込みシステムの多くにおいて，動作させるプログラム（タスク）が設計時に決まるという特性に着目して，以下のようなくふうを行っています．

▶ アドレス変換を行わないこと

汎用OSのメモリ保護機能は，それぞれのアプリケーションソフトウェア（プロセス）に独立したメモリ空間を提供することで，他のプロセスの使用するメモリ領域を見えなくしてしまうという考え方に基づいています．そのために，各プロセスごと

に仮想的なアドレス空間(論理アドレス)を用意し，メモリアクセスのたびにそれを実際のメモリアドレス(物理アドレス)に変換することが必要になります．

それに対して，動作させるプログラムが設計時に決まる場合には，それぞれのプログラムに独立したメモリ空間を与える必要性がなく，オーバヘッドをともなうアドレス変換は行わないほうが効率的になります．

▶メモリ配置を設計時に決定すること

汎用OSでは，OSを起動すると必要なプロセスを順に起動していきますが，プログラムを実際にメモリのどこに配置するかは，この過程で動的に決定します．

それに対して，動作させるプログラムが設計時に決まるならば，どのプログラムをどこに配置するかを設計時に決定したほうがメモリ使用量を抑えることができ，システムの起動にかかる時間を短くする効果もあります(これは，組み込みシステムにとっては重要な要件)．もちろん，メモリ配置を手動で決定する方法では，プログラムを修正するたびにメモリ配置の見直しや，メモリ保護のための情報の作り直しが必要となり，開発工数を上げてしまいます．

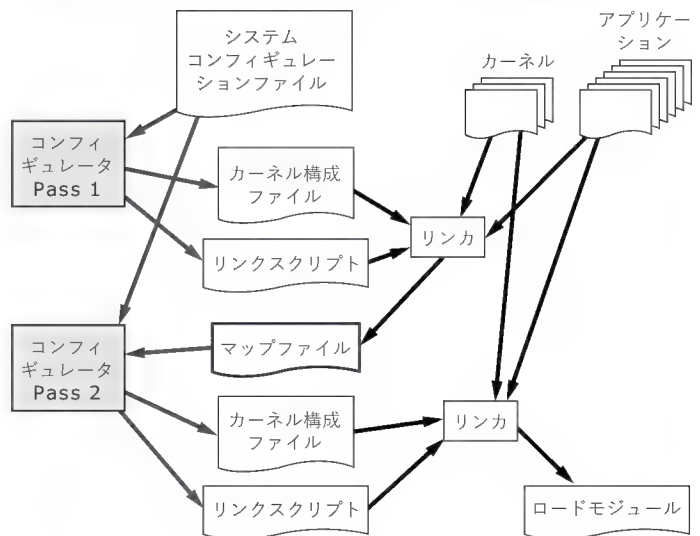
そこでIIMPカーネルでは，コンフィギュレータ(RTOSの構成を決定するためのツール)によってメモリ配置を決定するとともに，メモリ保護のための情報もコンフィギュレータによって生成します．これを実現するために，IIMPカーネルのコンフィギュレータは，**図2**に示す流れで処理を行います．まず最初のパスでコンフィギュレータは，仮のメモリ配置でRTOSの構成を決定します．これを仮にリンクし，このときに出力されるメモリマップ情報を参照して，第2のパスでメモリ配置を決定します．また，メモリマップや保護情報を決定するのに十分な情報が記述できるように，システムコンフィギュレーションファイルの記法も拡張されています．

TOPPERSプロジェクトがめざす「ITRONの良さを継承し，組み込みシステムの要求に合致したRTOS技術」が意味するところが，これらのIIMPカーネルの特徴から理解していただけるのではないかと思います．

なお，IIMPカーネルの開発は2002年3月に完了し，同年の6月からはトロン協会のWebサイトで配布されています．現在のバージョンは，ターゲットプロセッサとして，SH-3，i386(保護モード)，ARM940T(Memory Protection Unitをもったタイプの ARM)の3種類のプロセッサをサポートしています．

● IDLカーネル

OSが保護機能をもつことにより，あるアプリケーションのバグが，OSや他のアプリケーションに波及するのを防ぐことができますが，バグ自身がなくなるわけではありません．ソフトウェアを組み込んだ機器を出荷した後にバグが発見された場合，従来は，機器を回収(リコール)して修正する必要がありました．いうまでもなく，これには大きなコストを要しますが，この問題は，組み込みソフトウェアをネットワーク経由で簡単

〔図2〕IIMPカーネルにおけるコンフィギュレータの処理の流れ

にバージョンアップできるしくみがあると，かなり軽減することができます．

このことから，エーアイコーポレーションを代表とする企業などのグループによって，組み込みソフトウェアのバージョンアップ機能をもったμITRON仕様OSの開発を，IPAによる重点領域情報技術開発支援事業の採択テーマとして行いました．

具体的には，モジュールのダイナミックローディング機能を実現するために，

(1) IIMPカーネルをベースに，オブジェクトの動的生成機能を追加したリアルカーネル
(2) ターゲットシステム上で動作して，モジュールを動的にロードする機能をもつローディングエージェント
(3) サーバ上で動作して，ダウンロードするモジュールを用意するローディングサーバ
(4) ロードモジュールを作成するためのコンフィギュレータ

などを開発しました．

これらの開発成果を総称して，IDLカーネルと呼んでいます．IDLは，“ITRON with Dynamic Loading”の略です．IDLカーネルの詳細については，姉妹誌『デザインウェーブマガジン』2003年5月号 別冊付録「ITRONプログラミング・ガイド」に解説記事が載っているので，そちらを参照ください．IDLカーネルの開発は，2003年2月に完了しており，近日中にその大部分を公開する予定です．

TOPPERSライセンス

オープンソースソフトウェアを利用する上で，非常に重要になるのが，利用条件(ライセンス)です．

オープンソースソフトウェアのライセンスには，各種のもの

がありますが，もっとも広く使われているものが，GNUの開発ツールやLinuxに採用されているGNU一般公有使用許諾書(GNU General Public License，GNU GPL)と呼ばれているものです．GNU GPLの詳細についてはほかに譲りますが，組み込みシステムに用いるには，かなりの注意を要するものとなっています．具体的には，自分で開発したソフトウェアをGNU GPLのソフトウェアと一緒にリンクして利用した場合，自分で開発したソフトウェアもGNU GPLの条件で公開することを義務付けられるため，知的財産権が流出しないように十分な注意を払うことが必要です．

逆に，どのように使っても自由であるというライセンスにした場合，広く活用されていたとしてもそれを把握できず，開発成果の有効性を十分にアピールできないという問題があります．TOPPERSプロジェクトの開発成果の多くの部分は，国立大学の研究室やIPAの事業で開発したもので，国の予算(つまり税金)を使わせていただいて開発したことになります．国の予算を使って開発した以上，それによりどのような成果が上がったかを説明することが必要です．また，開発成果をアピールすることが，次の予算獲得，ひいてはプロジェクトの発展につながります．

以上のことから，TOPPERSプロジェクトで開発したソフトウェアには，GNU GPLなどの既存の利用条件を適用するのではなく，TOPPERSライセンスと呼ばれる独自の利用条件を設定することにしています．TOPPERSライセンスの全文を**図3**に示しますが，この中でもっとも特徴的なのは，条件(3)の(b)です．この条項により，TOPPERSのソフトウェアを機器に組み込んで利用する場合には，そのことをプロジェクトに報告するだけでよいことになります．われわれは，この考え方を「レポートウェア」と呼んでいます．

図3に示したTOPPERSライセンスの全文を読んでいただくと，GNU GPLとは違うといったにもかかわらず，GNU GPLが引用されている点が気になる方がいると思います．

具体的には，独自の利用条件〔条件(1)～(4)〕かGNU GPLのいずれかの条件に従うかを，ユーザーが選択できることになっています．これは，独自の利用条件を設定したソフトウェアをGNU GPLのソフトウェアと一緒にリンクすることを許すために広く使われている方法で，デュアルライセンスと呼ばれています．

前述したとおり，GNU GPLのソフトウェアと一緒にリンクする形で利用したソフトウェアは，GNU GPLの条件で公開することを義務付けられます．GNU GPLは注意を要すると書きましたが，GNU GPLのソフトウェアで有用なものは数多くあります．それらのソフトウェアをTOPPERSのソフトウェアとリンクして利用することができるように，GNU GPLも選択できることにしています．もちろん，どちらのライセンスを選ぶかはユーザーの自由ですから，GNU GPLを避けたい場合には，GNU GPLを選ばなければよいわけです．GNU GPLのソフトウェアと組み合わせて使用しない限り，GNU GPLは関係してこないと考えていただいてけっこうです．

〔**図3**〕**TOPPERSライセンス**(改定版)

<ソフトウェアの名称>
Copyright (C) <開発年> by <著作権者1>
Copyright (C) <開発年> by <著作権者2>
…

上記著作権者は，以下の(1)～(4)の条件か，Free Software Foundationによって公表されているGNU General Public LicenseのVersion 2に記述されている条件を満たす場合に限り，本ソフトウェア(本ソフトウェアを改変したものを含む．以下同じ)を使用・複製・改変・再配布(以下，利用と呼ぶ)することを無償で許諾する．
(1)本ソフトウェアをソースコードの形で利用する場合には，上記の著作権表示，この利用条件および下記の無保証規定が，そのままの形でソースコード中に含まれていること．
(2)本ソフトウェアを，ライブラリ形式など，他のソフトウェア開発に使用できる形で再配布する場合には，再配布に伴うドキュメント(利用者マニュアルなど)に，上記の著作権表示，この利用条件および下記の無保証規定を掲載すること．
(3)本ソフトウェアを，機器に組み込むなど，他のソフトウェア開発に使用できない形で再配布する場合には，次のいずれかの条件を満たすこと．
(a) 再配布に伴うドキュメント(利用者マニュアルなど)に，上記の著作権表示，この利用条件および下記の無保証規定を掲載すること．
(b) 再配布の形態を，別に定める方法によって，TOPPERSプロジェクトに報告すること．
(4)本ソフトウェアの利用により直接的または間接的に生じるいかなる損害からも，上記著作権者およびTOPPERSプロジェクトを免責すること．

本ソフトウェアは，無保証で提供されているものである．上記著作権者およびTOPPERSプロジェクトは，本ソフトウェアに関して，その適用可能性も含めて，いかなる保証も行わない．また，本ソフトウェアの利用により直接的または間接的に生じたいかなる損害に関しても，その責任を負わない．

注：TOPPERSライセンスは何度か改定しており，ここに示したものは，今後公開するソフトウェアに適用される最新版．これまで公開したソフトウェアには，古いバージョンの利用条件が適用されているが，基本的な考え方に違いはない．

進行中の開発項目と今後の開発計画

TOPPERSプロジェクト関連で進行中の開発項目を**表2**にリストアップします(一部計画中のものを含む)．JSPカーネルを各種のプロセッサへポーティングすることはもちろん，μITRON4.0仕様の他のプロファイル準拠のカーネルの開発や，JSPカーネル上で動作するソフトウェア部品の開発，ソフトウェア開発環境の整備など，広範な分野でソフトウェア開発を進めています．これらの中には，すでにほぼ完成しており間もなく公開できるものもある一方で，まだ開発の初期段階のものもありますが，実用化できる完成度になった時点で，オープンソースソフトウェアとして公開する予定です．

この中で，μITRON4.0仕様のほかのプロファイルのカーネルを開発するのは，各規模の組み込みシステムに対応するためです．開発にあたっては，一つのソースコードで各プロファイ

ルのカーネルを構成できるようにするのではなく，各プロファイルのカーネルを別々のソースコードで実現し，カーネルをシリーズ展開するアプローチをとります．これは，プロファイルごとに最適なカーネルの内部構造が大きく異なるため，前者のアプローチではJSPカーネルの「読みやすく改造しやすいソースコード」という特徴が損なわれるためです．

一例ですが，JSPカーネルではTCBは構造体の配列で実現していますが，自動車制御プロファイルのカーネルでは，構造体のフィールドごとに独立した配列としています．これは，(キャッシュをもつプロセッサでは)前者の方法のほうが実行効率が高いと考えられるのに対して，後者の方法のほうがRAM使用量を減らせるためです．

表2に挙げた開発項目の半数程度は，経済産業省による平成14年度即効型地域新生コンソーシアム研究開発事業(委託元：東北経済産業局)の採択テーマの一つである「組み込みシステムオープンプラットホームの構築とその実用化開発」の一環で開発しているものです．この研究開発事業では，筆者が統括リーダとなり，1大学・2高専・4公設試験所・6企業からなるコンソーシアムで，JSPカーネルの各種のプロセッサへのポーティング，ソフトウェア開発環境の整備，ソフトウェア部品の開発，製品への適用などの研究開発を進めています．

図4には，今後のカーネル開発のロードマップを示します．JSPカーネルから出発して，これまで開発してきたカーネル，現在開発中のカーネル，今後開発する予定のカーネルや技術の位置付けを示しています．このロードマップの中で，右上方向へ向かう開発は，ますます複雑化・大規模化する組み込みシステムに対応するための開発であると位置付けています．それに対し，右下方向へ向かう開発は，RTOS技術をシステムLSIへ組み込まれるソフトウェアに適用するための開発と位置付けています．

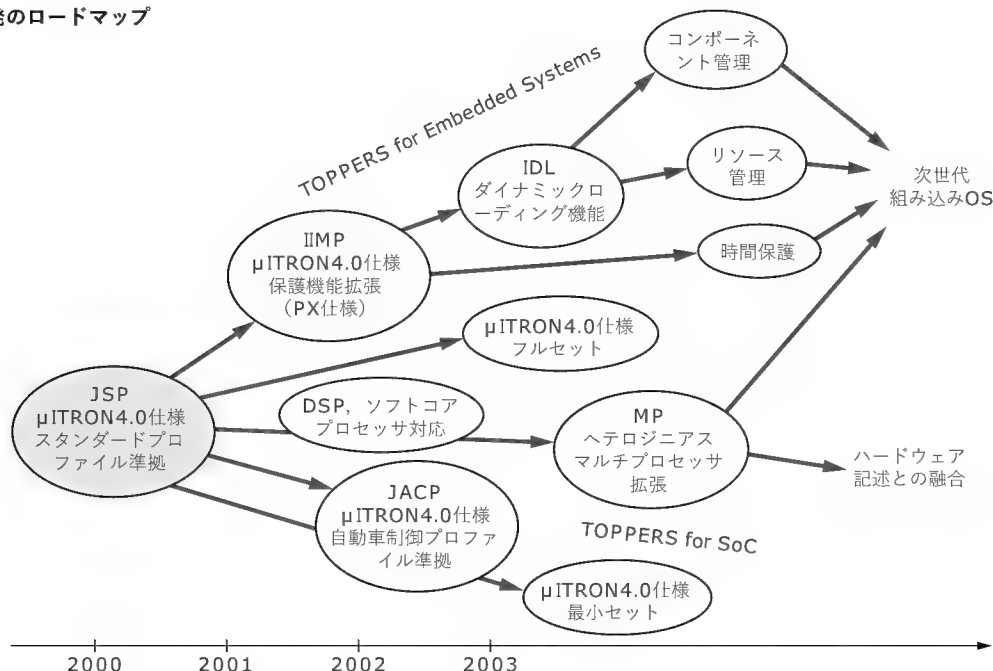
〔表2〕進行中の開発項目

開発項目
● JSPカーネルの各種のプロセッサへのポーティング ―― MIPS系，M16C，Nios，PowerPC系など
● μITRON4.0仕様の他のプロファイル準拠のカーネルの開発 ―― μITRON4.0仕様フルセット ―― 自動車制御用プロファイル ―― μITRON4.0仕様最小セット
●組み込みシステム向けのコンパクトなTCP/IPプロトコルスタック
● ITRONデバッギングインタフェース仕様への対応
●各種のデバイスドライバやライブラリ
●マルチプロセッサ対応のカーネル
●“Linux on ITRON”(JSPカーネルとLinuxのハイブリッドOS)

TOPPERSプロジェクト発展の方向性

これまで説明してきたとおり，TOPPERSプロジェクトでは，「オープンソース」を，組み込みシステム技術の発展と産業の振興のための「手段」であると考えています．オープンソースは，決して「目的」ではないのです．

TOPPERSの開発成果を継続的に保守・発展させ，独自の組み込みシステム技術を開発していくためには，プロジェクトの参加企業がTOPPERSやその周辺のソフトウェア開発に対して研究開発投資する状況を作らなくてはなりません．その大前提が，TOPPERSの周辺で参加企業のビジネスが成立する

〔図4〕カーネル開発のロードマップ

ことです．

参加企業のビジネスを成立させるためには，参加企業がTOPPERSプロジェクトの成果を利用して開発したソフトウェアを，すべてオープンソース化しなさいというわけにはいかないと考えています[注3]．たとえば，JSPカーネル上で動作するソフトウェア部品を開発して販売してよいのはもちろん，JSPカーネルを独自に機能拡張したものをオープンソース化せずに販売することも許されます（ただし，この場合TOPPERSライセンスは守らなければならない）．

実際すでに，TOPPERSプロジェクトの参加企業は，それぞれのビジネスにプロジェクトの成果を活用し始めています．JSPカーネルに対するサポートサービス（ポーティングの請負や技術サポート，バグフィックスを約束する保証サービスなど）を提供している会社はすでに複数ありますし，自社のソフトウェア開発環境やマイコンボードに，JSPカーネルをバンドルしている会社もあります．また，JSPカーネルを利用した製品を開発・販売している会社もあります．

いうまでもないとは思いますが，TOPPERSプロジェクトが，参加企業のビジネスモデルを設計したり，調整したりすることはありません．あくまでも，一定のルールの元で，参加企業が自由にビジネスモデルをくふうすることになります．この点は，多くの企業が自由にビジネスモデルを構築しているLinuxが良いモデルです．ここでいう「一定のルール」とは，まさにオープンソースソフトウェアの利用条件（ライセンス）がそれに該当します．つまりTOPPERSにおいては，Linuxとは異なる「一定のルール」を設定しているわけなので，可能なビジネスモデルもLinuxとは異なるものになるだろうと考えています．

幸いなことに，組み込みシステム分野においては，オープンソースソフトウェアをベースにしたビジネスが成立しやすいと思われます．これは，組み込みシステム分野においては，システムごとの要求事項やプラットホーム（プロセッサや開発環境）が多様であり，標準的なソフトウェアがオープンソースで存在しても，それを要求事項に合致するように改造したり，独自のプラットホームにポーティングするといった要求がつねに存在するためです．

TOPPERSプロジェクト発展の方向性としてもう一つ重視しているのが，組み込みシステム技術者の育成です．組み込みシステム業界がかかえる大きな課題の一つは，開発技術者の質・量両面での充実が必要であるにもかかわらず，技術者の育成のための良い教育の場や題材が十分に提供されていないことです．

この方面では，組み込みソフトウェア管理者・技術者育成研究会（SESSAME，リーダ：飯塚悦功 東京大学教授）が，多くのボランティアの力により良い成果を挙げつつあり，TOPPERSプロジェクトもこれに連携して活動を進めています．具体的には，前にも述べたとおり，TOPPERSのオープンソースソフトウェアを教材として整備し，それを活用したセミナの開催や参考書の作成を考えています（この連載も，そのための活動の一つ）．また，TOPPERSプロジェクトの参加メンバには，公設試験所（各県の工業試験所など）や大学・高専が含まれており，すでにJSPカーネルを教材として活用している例もあります．それらを教育の場として活用することもできると考えています．

最後に，プロジェクトの開発成果の海外展開にも，今後力を入れていきたいと考えています．海外展開にあたっては，アジア諸国（とくに中国，台湾，韓国）への展開に重点をおいて取り組んでいく予定です．

おわりに

本稿では，連載「TOPPERSで学ぶRTOS技術」の第1回として，TOPPERSプロジェクトの概要と展開について紹介しました．次回以降では，今回紹介したソフトウェアの活用技法の紹介や，開発成果の技術解説を行っていきます．このような話題を取り上げてほしいといった要望や，ITRON仕様やTOPPERSに関する質問があったら，連載の中で取り上げていきたいと考えているので，編集部までお寄せいただけると幸いです．

たかだ・ひろあき
TOPPERSプロジェクト会長，名古屋大学大学院 情報科学研究科

注3：この点でも，オープンソースソフトウェアの派生物はすべてオープンソースでなければならないとするGNU GPLとは考え方が異なっている．もちろん，プロジェクトの一環として開発したソフトウェアは，TOPPERSライセンスによりオープンソース化するのが原則だが，プロジェクトの成果を利用していても，企業が独自に開発したソフトウェアには，この原則は適用されない．

マルチデバイス/マルチコア開発に対応した

JTAGデバッグツール「WIND POWER *ICE/IDE*」の概要

福徳信夫

はじめに

ブロードバンドが急速に普及し，私たちの生活に新しい電子機器やサービスが導入されてきた．携帯電話での写真や映像の撮影と送付，ゲームならびに映画のオンデマンドによる配信も始まりつつある．こうしたインフラを支える機器は，高速処理と高信頼性を要求される．たとえば，Gigabit Ethernetに対応した機器などの需要も高まっている．

たとえば，前述したGigabit Ethernetだと，大量のデータがnsという時間で転送されてくる．既存のRISCプロセッサとRTOSを利用した機器でも，プロセッサのデータ処理時間はμsのレベルになる．これでは，高速に転送されてくるデータを処理できないことになってしまう．処理できたとしても，コントロール用のコンソールなどの別処理が遅くなってしまう可能性もある．そのため最近の電子機器では，コンソールを管理するプロセッサならびに通信を管理する専用ハードウェア（プロセッサ，DSP，FPGA）などのマルチデバイスで構成される機器開発が行われている事例も見受けられる．さらに，機器の低消費電力化ならびに低価格化の要求に複数のコアをSoC（システムオンチップ）に搭載している事例もある．

このような複雑なシステムを短期に開発することが，機器開発者に要求されている．もちろん高信頼性が要求される．そこで，マルチデバイス，マルチコアのシステムの開発に要求される要件，開発環境，ツールについて解説する．

JTAG（Joint Test Action Group）とJTAGデバッグツール

JTAG（**図1**）はIEEE1149.1で定められた規格である．この規格のもともとの目的は，ボード上に実装されたデバイス，ならびに半導体内部をデイジチェーン接続し，TDI（テストデータ入力）からデータを送信してTDO（テストデータ出力）から得られる結果によってボードならびに半導体のテストを行うハードウェアテスト規格だった．TDI，TDO以外の信号としてはTCK（クロック），TMS（内部ステータス），TRST（リセット）信号などから構成される．

本来JTAGは，ボードテスト，半導体テスト用機器に使用されるのが一般的だった．高機能化により100MHzを超える外部クロックで駆動されるプロセッサも普及している今日，従来のICE（インサーキットエミュレータ）形式のデバッグツールは開発しにくく，JTAG方式のデバッグツールが一般化しつつある．そこで，各半導体メーカーはJTAG規格を拡張し，チップ内部にOCD（オンチップデバッグ）機能を搭載し，たとえばTDIからメモリ値を読むコマンドを入力するとTDOからその値を読み出すような機能を追加している．各社で実装手法が異なり，JTAGハードウェア規格が同一であっても，OCDのコマンド体系は統一されていないのが現状である．

JTAGデバッグツールとは，このJTAG規格に基づいて各半導体メーカーが実装しているOCDのコマンドを入出力してプロセッサのデバッグを行う機器をいう．ホスト（PCなど）とツールはEthernetなどのインターフェースで接続されている．ホスト上にはC/C++コンパイラから生成されるプログラムを開発機器のメモリにダウンロードする機能，C/C++デバッグ，メモリ表示変更，レジスタ表示変更機能などをサポートするデバッガがJTAGデバッグツールをもコントロールする．

JTAGデバッグツールがデバッグするマルチデバイス/マルチコアとは？

マルチデバイス/マルチコアの定義は，「複数のボードで機器が構成され，各ボードにはプロセッサが実装されている」というものである．**図2**(**a**)では，JTAGスキャンチェーンは各ボード

〔**図1**〕**JTAG**

内で接続されており，各ボードのJTAG回路を外部で接続してデバッグ対象となるプロセッサを接続する．**図2(b)**は，一枚のボード上に複数のプロセッサが実装されJTAGで接続されている．**図2(c)**は一つのシリコン上に複数のプロセッサコアが実装され，それぞれがJTAGで接続されている．ここでデバッグ対象はプロセッサということで限定しているがDSP，FPGAあるいはカスタムプロセッサなどでJTAG接続できる半導体も含まれる．

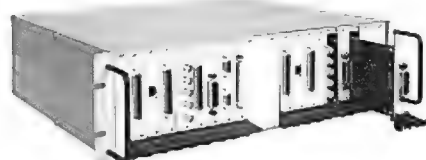
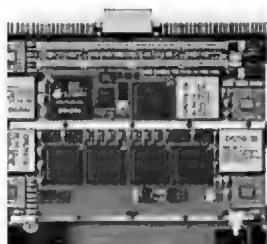
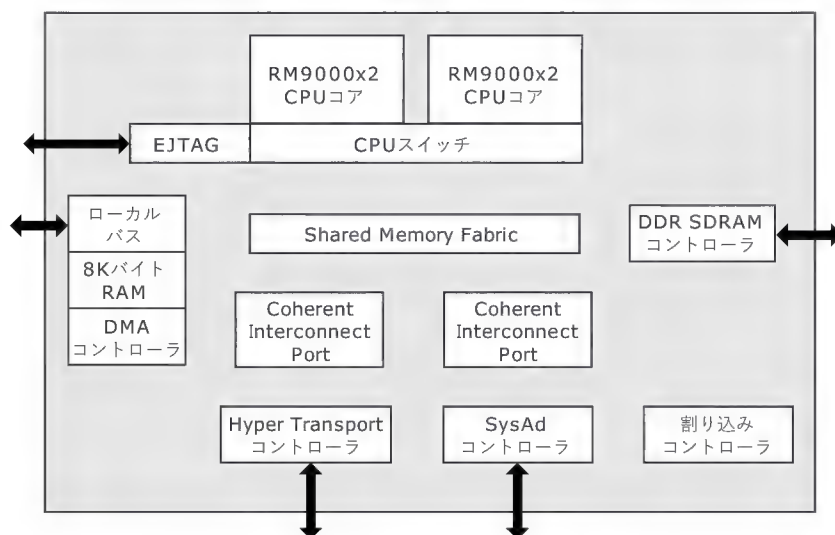
マルチプロセッサを搭載した回路の設計基準

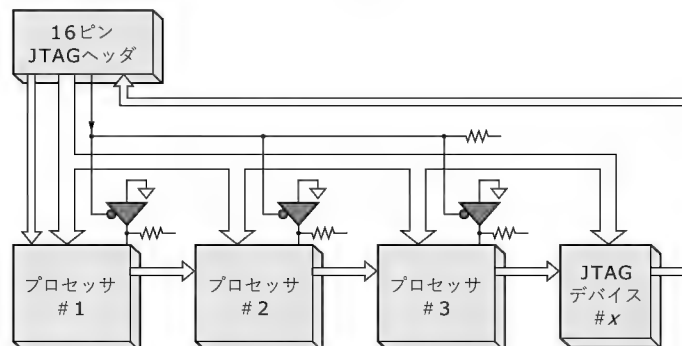
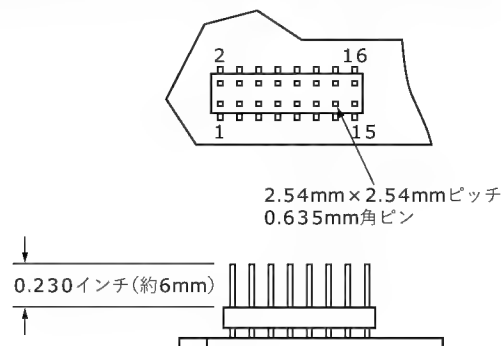
マルチプロセッサ/デバイスを実装する際の考慮事項を解説する．なお以降の説明では，マルチプロセッサ/デバイスのICE「WIND POWER *ICE*」(ウインドリバー社)を例にして解説する．**図3**に複数のプロセッサを接続した例を示した．

図4にJTAGデバッグツールを接続するコネクタの仕様を示す．16ピンの一般的に市販されているコネクタを使っている．信号名の詳細を**表1**に示す．TDO，TDI，TCK，TRST，TMSはJTAGの標準信号で，HRESETはプロセッサ特有のリセット信号である．VDOに関しては，プロセッサの電源に接続する必要がある．**表2**にWIND POWER *ICE*の電気仕様を示す．

● 回路設計の注意点

- JTAGデバイスは可能なかぎりJTAGコネクタに近接して配置する．JTAG信号(TDO，TDI，TCK，TMS，TRST)の配線も適切に配置することが必要である
- TCK，TMSは，タイミングが変わらないように同一の距離で配線する
- TDO信号は最後のデバイスからJTAGコネクタに接続されるTDO信号にJTAGコネクタに近接して反射をおさえるためにダンピング抵抗を挿入する
- マルチデバイス設計で重要なポイントは，TCK(クロック)はJTAGデバッグツールから供給され接続されている全デバイスに供給されるSOCの場合はクロックnのデバイス間の配線長もあまり長くならないので問題になる場合は少ないが，複

〔図2〕マルチデバイス/マルチコアの定義

(a) 複数CPUボードのデバッグ

(b) ボード上の複数CPUのデバッグ

(c) シリコン上の複数CPUのデバッグ

〔図3〕回路接続例

〔図4〕JTAGデバッグツールを接続するコネクタの仕様

数のボード，ボード上に複数デバイスが搭載されている場合は，クロック信号の劣化，タイミングが変更されないように考慮する必要がある

マルチデバイス/マルチコア対応 JTAGデバッグツール「WIND POWER *ICE*」の概要

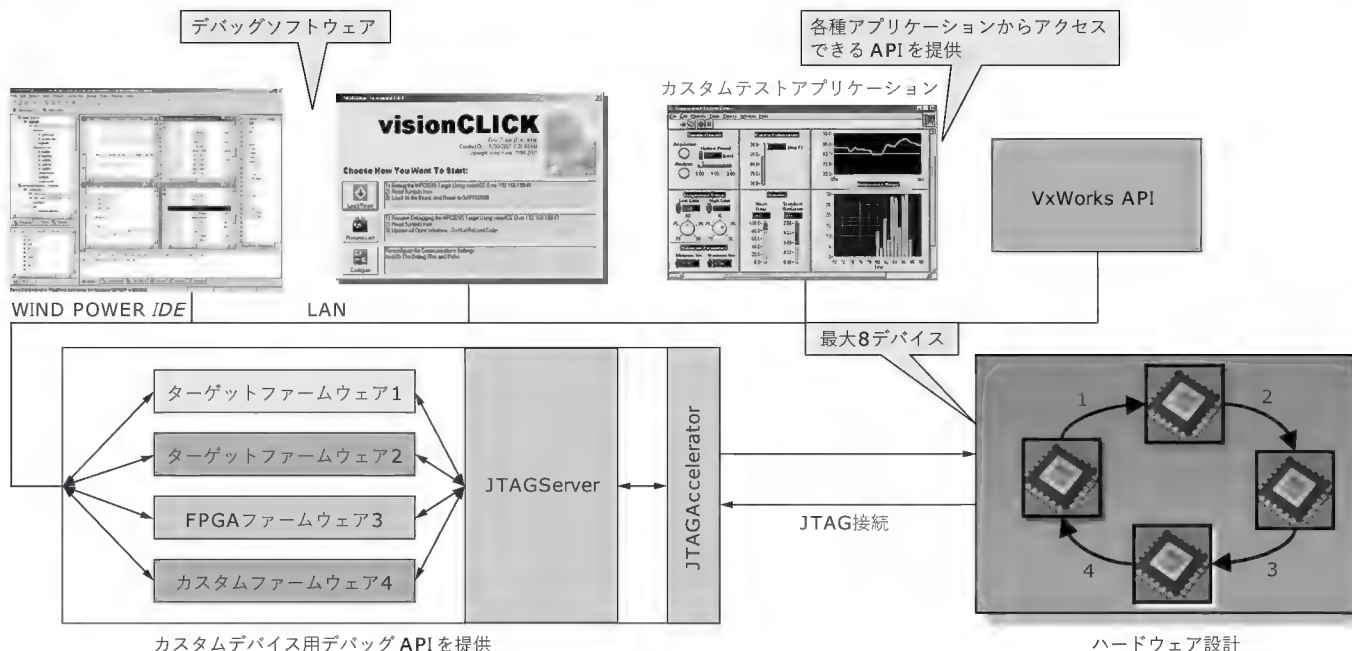
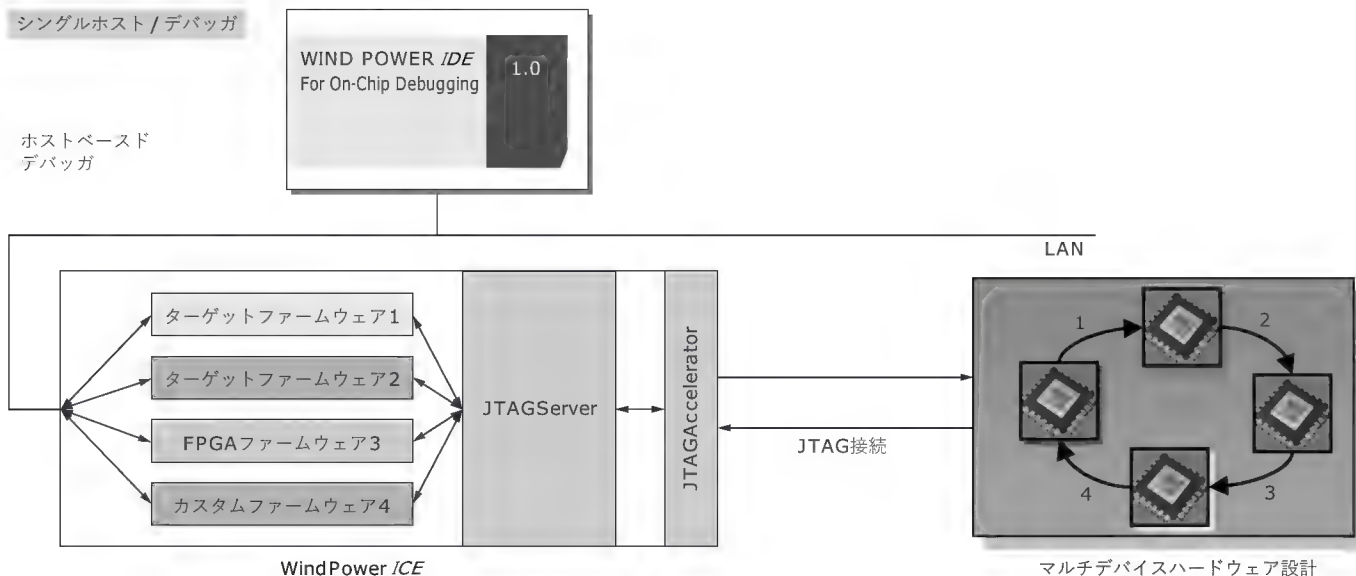
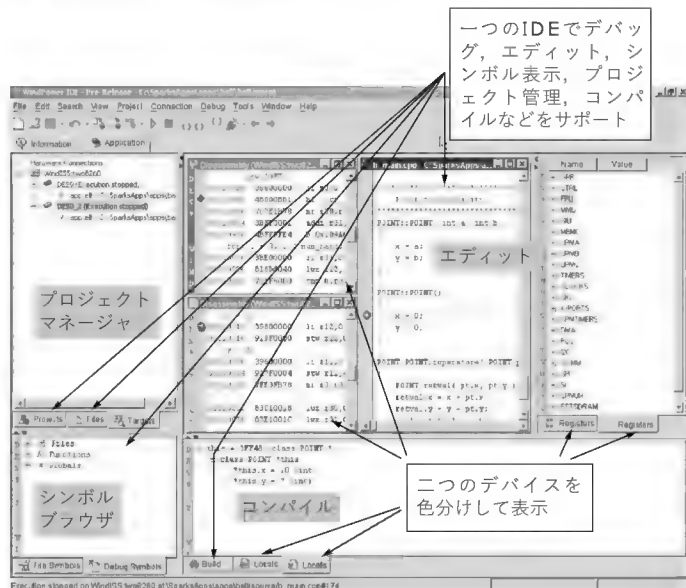
既存の一般的に普及しているJTAGデバッグツールは単一のプロセッサ，デバイスのみに接続してソフトウェアデバッグを行う設計だった．つまり，マルチデバイスで設計された回路でもJTAGチェーンを分離して個々のデバイスをデバッグする必要があった．そのため製品開発サイクルの短縮，開発コスト低減に結びつかない場合も散見された．この状況を配慮して設計開発されたマルチデバイス/マルチコアを1台のJTAGデバッグツールでサポートするツール「WIND POWER *ICE*」の概要を解説する．**図5**に「WIND POWER *ICE*」の概念図を示した．**図5**の右下部分はデバッグするボード，SoCを示している．各デバイス間はJTAG（TDO，TDI，TCK，TMS，TRST）で接続されている．**図5**の左下はマルチデバイス，コア対応のJTAGデバッグツールの模式図である．

〔表1〕信号名の詳細

ピン番号	信号名（note）	機　能
1	TDO	テストデータ出力
2	N/C	(Not connected)
3	TDI	テストデータ入力
4	TRST	テストリセット
5	N/C	No Connect
6	V_{DD}	プロセッサ電圧
7	TCK	テストクロック入力
8	N/C	No Connect
9	TMS	テストモード選択
10	N/C	No Connect
11	N/C	No Connect
12	N/C	No Connect
13	HRESET	ハードウェアリセット
14	N/C	No Connect
15	N/C	No Connect
16	GND	DC Ground

〔表2〕WIND POWER *ICE*の電気仕様

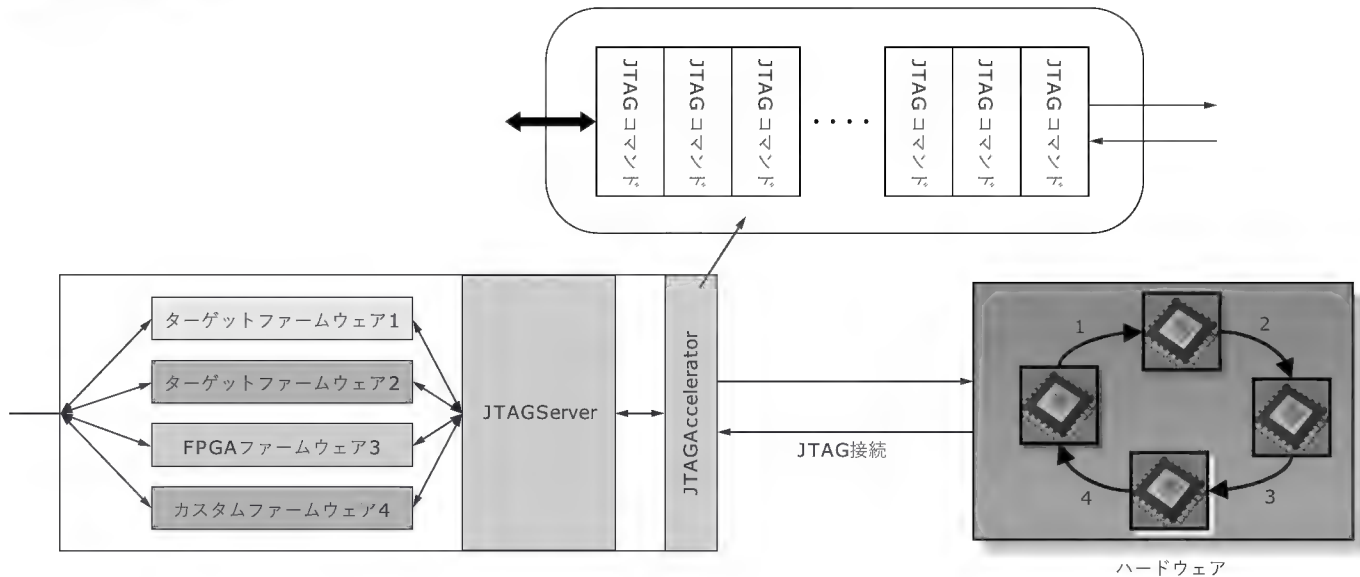
V_{IO}	1.8V		2.5V		3.3V	
	MIN	MAX	MIN	MAX	MIN	MAX
V_{IL}		0.63V		0.7V		0.8V
V_{IH}	1.17V		1.7V		2.0V	
I_{OL}	12mA		20mA		24mA	
I_{OH}	− 12mA		− 20mA		− 24mA	
I_{OL} TCK	24mA		40mA		48mA	
I_{OH} TCK	− 24mA		− 40mA		− 48mA	
V_{OL}		0.3V @ 8mA		0.4V @ 16mA		0.5V @ 24mA
V_{OH}	1.35V @ 8mA		1.85V @ 16mA		2.5V @ 24mA	
V_{OL} TCK		0.3V @ 16mA		0.4V @ 32mA		0.5V @ 48mA
V_{OH} TCK	1.35V @ 16mA		1.85V @ 32mA		2.5V @ 48mA	
C_{IN}	10pF					
C_{OUT}	100pF					

〔図5〕WIND POWER *ICE*の概念図

〔図6〕シングルデバッガ/マルチデバイス対応

〔図7〕異なるデバイスの表示のようす

また，最大8デバイスまで同時に同期してデバッグできる．内部にデバッグ対象のそれぞれのデバイスに対応したコントロールソフトウェア（ファームウェア）を最大八つまで搭載できる．1番目のファームウェアは一つ目のデバイスをコントロール，2番目は二つ目のデバイスを独立してコントロールする．このファームウェアはウインドリバー社が標準で提供するプロセッサ，FPGAなどに対応できる．また，カスタム設計のデバイスに対してはAPIを公開するので，ユーザー自身でファームウェアを設計できる機能ももっている．JTAGチェーンは単方向でシリアル接続になっているため，マルチデバイスをコントロールするために並列にある内部ファームウェアの制御をシリアルに変換する必要がある．その制御を行うのがJTAGServerと呼ばれる機能である．JTAGServerは，対象デバイスを適宜選択したファームウェアと接続する機能をJTAGデバッグツールに与える．また，一般的にデバイス内のJTAGアクセスは1万ビットにもおよぶレジスタをスキャンする必要があり，時間がかかってデバッグの応答時間に影響を与える場合がある．そのため，JTAGアクセスを高速化する手段としてJTAGAcceleratorを開発し，スピードアップをはかっている．

マルチデバイス/マルチコア対応JTAGデバッグツールの要件

マルチデバイスをデバッグするとき，次の二つの場合を考慮する必要がある．一つ目は，一人のユーザーが1台のホストを使用してマルチデバイスをコントロールすることが考えられる．**図6**の状況である．マルチデバイスをコントロールする際，ヒューマンエラーを排除する考慮が必要である．たとえば，異なるデバイスのメモリを表示する上で，デバイスごとに各メモリウィンドウを色分けして表示している（**図7**）．マウスカーソルをウィンドウに移動すると，そのデバイスを色分けして識別するようにGUIを設計した．各デバッグ用のボタン（実行，停止，シングルステップなど）も，マウスを移動したときに色が変わり，現在，どのデバイスをデバッグしているのかを簡単に識別できるようにくふうした．

二つ目は，複数のユーザーがそれぞれのデバイスにアクセスする必要がある場合である．マルチデバイス対応JTAGツールは，複数ホストからアクセスできる機能が要求される．ウインドリバー社のWIND POWER *IDE*に含まれるデバッガは，こ

〔図8〕JTAGAccerelator

の二つの要件に合致するように設計されている．インストラクションセットシミュレータも搭載され，ハードウェア完成前のプログラムのシミュレーションへの配慮をしている．

JTAGServer機能と，JTAGアクセスを高速化する技術「JTAGAccerelator」について

JTAGServerは，JTAG規格に準拠したハードウェア規格に対応し，マルチデバイス/マルチコアのソフトウェアデバッグインターフェースを提供する．スキャンチェーン上にあるデバイスの順番，デバイスIDなどを定義，認識する．スキャンチェーンにシリアル接続されるデバイスにコマンドを送信するうえで必要な機能をサポートする．たとえば，同一アーキテクチャのデバイスが複数接続されている場合に何番目のデバイスにアクセスするのかを識別する必要がある．

APIを公開してJTAGデバッグツールをコントロールするホスト上のソフトウェアの開発ができる．サードパーティのソフトウェア製品，カスタムソフトウェア製品と接続でき，低価格なJTAGを応用したボードテスタ，フラッシュライトシステムなどへの応用も可能である．最大128個のJTAGデバイスをスキャンチェーン上に接続可能で，その中から最大8デバイスを選択してソフトウェアデバッグができる．マルチデバイスでは，同期デバッグ(全デバイスを同時スタート，ストップ，初期化など)が必要であり，その機能もサポートしている．

TCK(JTAGクロック)を高速化せず，JTAGアクセスを高速化する手法としてJTAGAccerelatorが開発された．**図8**に示すように，JTAGコマンドを連続して格納できるバッファを用意し，JTAGアクセスの空き時間を最小にするようにくふうした．結果として約3倍程度のアクセスの高速化を実現した．

マルチデバイス/マルチコアのソフトウェアデバッグを実現したツール事例

ウインドリバー社のWIND POWER *ICE*はマルチデバイス，マルチコアを1台のツールで実現したJTAGデバッグツールである．PowerPC，MIPS，ARM/FPGA，カスタムデバイスのアーキテクチャに対応している．また，マルチデバイス，マルチコア用デバッガとしてWIND POWER *IDE*が提供され，単一ホストからマルチデバイスへのアクセス，複数ホストからそれぞれのマルチデバイスへアクセスする機能をもっている．

おわりに

われわれを取り巻くブロードバンド環境はますます高機能化，高性能化していく．その際に電子機器開発を支える技術の概要を解説した．製品開発エンジニアにとって，製品開発サイクルの短縮，製品開発コストが低減され，信頼性の高い電子機器開発が実現されることを希望する．

ふくとく・のぶお ウインドリバー(株)

ソフトウェアの部品化を容易にする

ITRON のソフトウェアグループ管理の概要

金田一勉

日本の組み込みシステムでは，その半数以上がITRON仕様API を採用しています(トロン協会アンケートより)．その反面，現状ではミドルウェアやソフトウェア部品の不足が問題視されています．

本稿では，ソフトウェアの部品化を容易にする機能である「ソフトウェアグループ管理」について解説します．この「ソフトウェアグループ管理」は，本誌2003年2月号の「保護機能をもったμITRON仕様準拠カーネル」でも概要を説明しました．「ソフトウェアグループ管理」(特許出願中)においては，保護機能は副産物であり，主目的は**ソフトウェアの部品化**にあります．

そこで本稿では，ソフトウェアの部品化を容易にする機能とその効果について，解説します．

ID番号の割り付け

ITRON仕様では，カーネルオブジェクト(以下オブジェクト)は，ID番号で管理します．オブジェクトに対しID番号を割り付ける方法は，以下の三つがあります．

① ユーザーがID番号を指定する(cre_xxx)
② カーネルが割り付けて通知する(acre_xxx)
③ システムコンフィギュレーションにより割り付ける(静的API CRE_XXX)

①は，オブジェクトの一つ一つにユーザーがID番号を割り付け，その番号をオブジェクトの生成要求時に指定する方法です(**図1**)．この方法では，オブジェクトが増えたときに，そのID番号の割り付けを行い，ソースファイルをコンパイルしなおす必要があります．通常，ユーザーが割り付けたID番号をC言語のヘッダファイルでシンボルに定義し，そのファイルをソースファイルでincludeして，定義したシンボルをオブジェクト生成時に指定します．

②は，オブジェクトの生成要求時に，カーネルにID番号を割り付けてもらう方法です(**図2**)．この方法では，カーネルから通知されるID番号を変数などに記憶します．オブジェクトに対しアクセスを要求する場合は，その変数からID番号を取得し，サービスコールに指定します．

③は，システムコンフィギュレーションファイルでシステムに組み入れるオブジェクトを定義する方法です(**図3**)．コンフィギュレータというツールがそのファイルを解析し，ID番号を定義したファイルを出力します．そのファイルをソースファイルでincludeして，オブジェクトに対するサービスコールでそのシンボルを指定します．この方法では，システムをコンフィギュレーションするたびに，ソースファイルをコンパイルしなおす必要があります．

「ソフトウェアグループ管理」は，①の方法を採用しているITRON仕様準拠カーネルへの機能導入に適しています．コンフィギュレータの対応により，③の方法を採用しているカーネルでも導入が可能です．

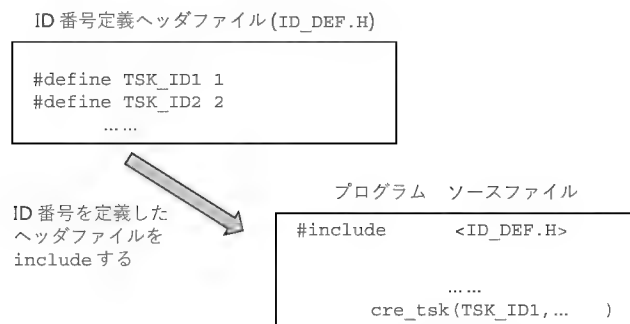
〔図1〕ユーザーが定義する方法

ID番号定義ヘッダファイル(ID_DEF.H)

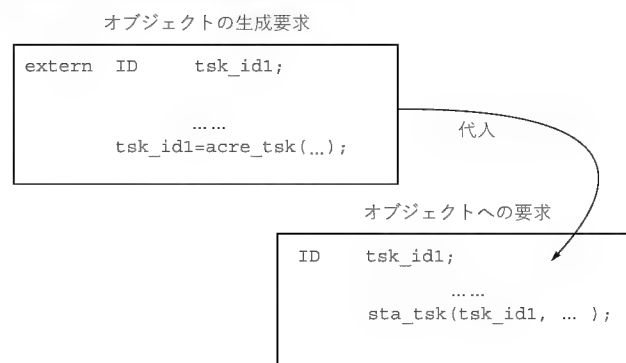
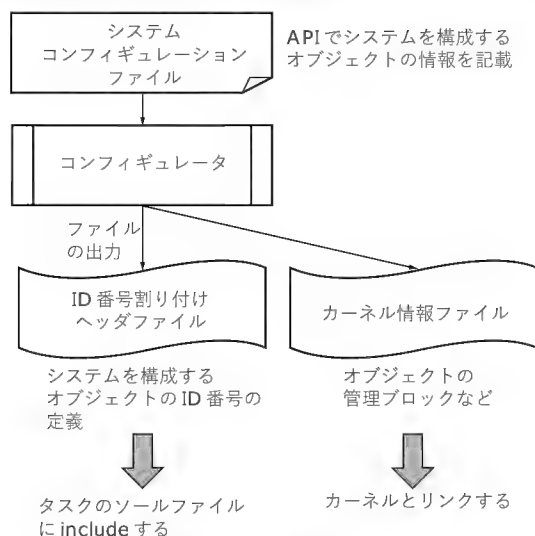
```
#define TSK_ID1 1
#define TSK_ID2 2
      ……
```

ID番号を定義したヘッダファイルをincludeする

プログラム　ソースファイル

```
#include       <ID_DEF.H>

                  ……
     cre_tsk(TSK_ID1,...    )
```

〔図2〕カーネルが割り付ける方法

〔図3〕コンフィギュレータを使う

〔図4〕従来のID番号管理

「ソフトウェアグループ管理」の概要

「ソフトウェアグループ管理」は，システムをいくつかのグループに分け，そのグループごとにID番号を管理するしくみです．

ID番号の管理は，二つあります．一つは，従来と同じように管理する方法で，「グローバルID番号管理」といいます．グローバルID番号は，どのプログラムからも指定ができます．もう一つは，グループごとにID番号を管理する方法で，「ローカルID番号管理」といいます．

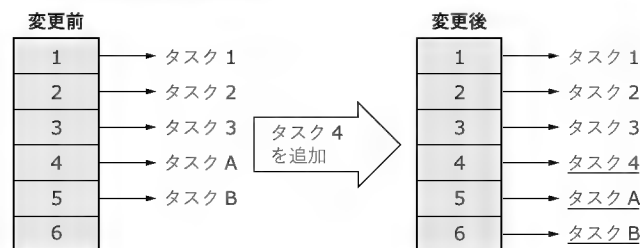
ローカルID番号は1番から始まるので，ID番号の割り付けは，システムに組み入れる際に割り付けるのではなく，グループ内で自由に割り付けを決めることができます．ローカルID番号を割り付けたオブジェクトは，他のグループからアクセスできません．

グループには，グローバルID番号を割り付けるオブジェクトを含めることができます．グローバルID番号を割り付けたのがタスクである場合，そのタスクは，グローバルID番号を割り付けたオブジェクトと，タスクが所属するグループにあるローカルID番号を割り付けたオブジェクトの両方にアクセスできます．

● 従来のID番号割り付け

オブジェクトのID番号は，通常1番から始まる番号を割り付けます．オブジェクトが追加される場合，未使用の番号を割り付けるか，ID番号を割り付けし直します．しかし，未使用の番号を割り付けていくと，オブジェクトの関連が番号から推測することが難しくなっていきます．

システムにタスクを追加する例を**図4**に示します．

システム(変更前)にタスク4を追加するとした場合，追加したタスク以降に割り付けしたタスクのID番号が変更になります(変更後)．

追加するタスクのID番号を決定し，追加した以降のID番号に割り付けるタスクのID番号も変更します．この場合，ID番号を使用するプログラムを再構築(コンパイル)する必要があります．

● 「ソフトウェアグループ管理」を導入した例

システムに二つのグループを登録し，グループ1にタスク4を追加する場合を**図5**(次頁)に説明します．

追加するタスク4のID番号を，グループ1のローカルID番号を割り付けた場合，その割り付けはグループ1だけに影響し，グローバルID番号や他グループのID番号の割り付けには影響しません．

そのため，グループ1のプログラムを再構築(コンパイル)するだけで，システムへの組み入れができます．

● オブジェクト保護

グループ内のローカルID番号を割り付けたオブジェクトは，そのグループに所属するタスクからしかアクセスができません．たとえば，「ソフトウェアグループID番号管理」の例で，タスク1がローカルID番号1番を指定した場合，タスク2を対象としたことになります．タスクAがローカルID番号1番を指定した場合，タスクBを対象としたことになります．

ローカルID番号は，所属するグループ内に割り付けているオブジェクトが対象になるので，他グループのローカルID番号を割り付けたオブジェクトを指定することはできません．

図5(**a**)におけるオブジェクトへのアクセス許可/不許可の表を**表1**に示します．

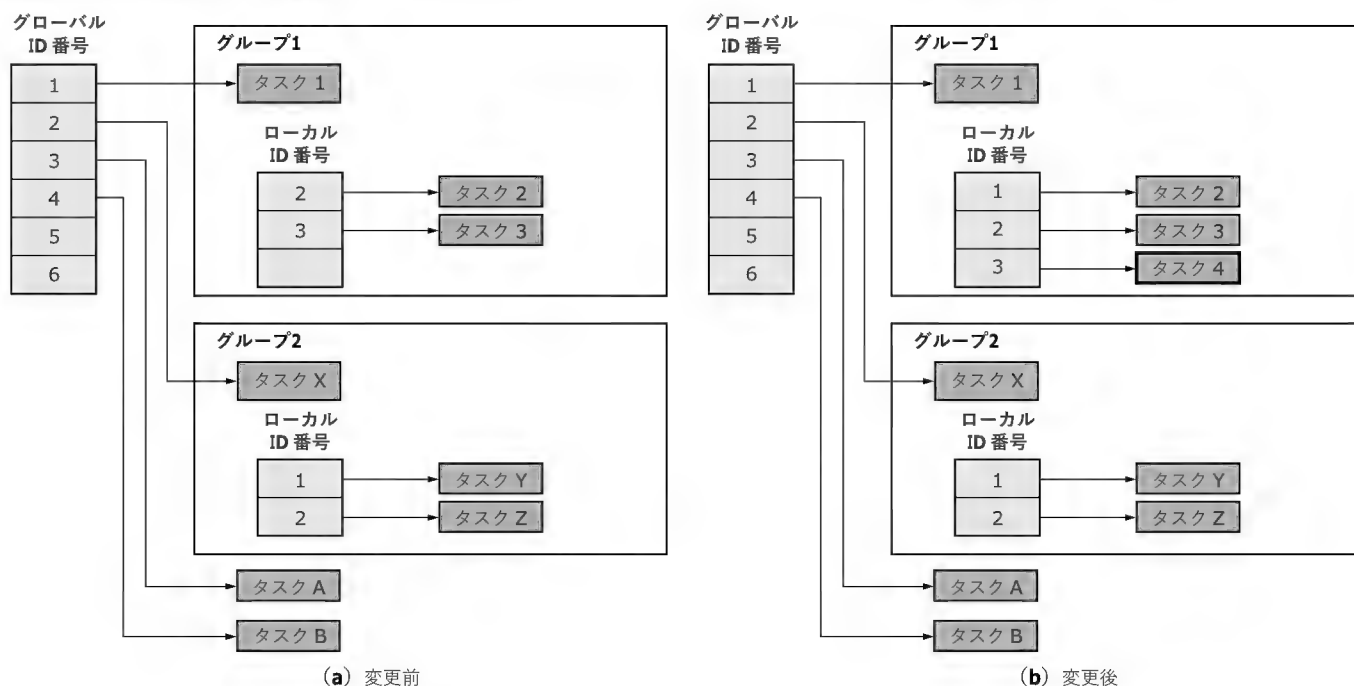
グローバルID番号を割り付けたオブジェクトは，どのタスクからもアクセスできますが，ローカルID番号を割り付けたオブジェクトは，それが所属するグループのタスクからのアクセスのみが許可されます．

● ID番号の指定方法

ID番号の指定は16ビットの正数とし，グローバルID番号は0001H～00FFHまで割り付けることができるとします．

ローカルID番号は，ID番号にローカルID番号を指定するマーク(0F00H)とOR(論理和)するものとします．つまり，ローカルID番号の指定としては，0F01H～0FFFHになります．

タスクのサービスコールでオブジェクトのID番号を指定する場合，グローバルID番号のオブジェクトかグループ内ロー

〔図5〕ソフトウェアグループ管理

〔表1〕アクセス許可/不許可表

アクセス対象 / アクセス側		グループ1			グループ2				
		タスク1	タスク2	タスク3	タスクX	タスクY	タスクZ	タスクA	タスクB
グループ1	タスク1	○	○	○	○	×	×	○	○
	タスク2	○	○	○	○	×	×	○	○
	タスク3	○	○	○	○	×	×	○	○
グループ2	タスクX	○	×	×	○	○	○	○	○
	タスクY	○	×	×	○	○	○	○	○
	タスクZ	○	×	×	○	○	○	○	○
	タスクA	○	×	×	○	×	×	○	○
	タスクB	○	×	×	○	×	×	○	○

カルID番号のオブジェクトを指定するかは，マークを付けるか/付けないかの指定になります．

従来との互換性

「ソフトウェアグループ管理」では，ID番号の指定により対象のオブジェクトの所属を指定する機能であるため，ITRON仕様で用意しているサービスコールのAPIに影響はありません．そのため，タスクのプログラムの修正がほとんど必要ありません．グローバルID番号は従来と同様の管理なので，「ソフトウェアグループ管理」機能を利用しなくとも，従来のままの利用も可能になります．

グループ番号を意識するのは，オブジェクトの登録時と非タスクコンテキスト部でのサービスコールでオブジェクトを指定する場合です．

非タスクコンテキスト部は，グループ内ローカルID番号を割り付けたオブジェクトの指定も可能としています．非タスクコンテキスト部がグループ内ローカルID番号を割り付けたオブジェクトを指定する場合は，前述したマーク(0F00H)をグループ番号(0100H～0E00H：グループ番号1～14に相当)に置き換えてORしたID番号を指定します．

導入の効果

システムに「ソフトウェアグループ管理」を導入することによる効果を記載します．

● 機能を利用するユーザー側

- ソフトウェアグループ管理は，サービスコールで指定するID番号を工夫しているだけなので，導入に際し，プログラムの変更が少ない．
- ソフトウェア部品をシステムに組み入れる場合，グループを構成するオブジェクトのID番号割り付けは，他グループと

の情報交換するオブジェクトだけになるため，ID 番号の割り付けが容易になる．

- グループを構成するオブジェクトが増えた場合，ID 番号を割り付け，グループを構成するプログラムのソースファイルを再度コンパイルするだけで済む．

　このため，オブジェクトの増減への影響を小さくすることができ，全ソースファイルの再コンパイルの手間を省くことができる．システムを構成するプログラムのすべてのソースファイルを管理する手間を省くということと，開発効率を向上させることができる．

- ソフトウェア部品の提供側
- ソフトウェア部品が内部で使用するオブジェクトの構成を隠ぺいできるので，機能変更などによるオブジェクト構成の変更が容易になる．
- 従来は，ソフトウェア部品をシステムに登録する際に，ソフトウェア部品を構成するオブジェクトに ID 番号を割り付ける必要があった．そのため，ソフトウェア部品を提供する側は，部品を構成するオブジェクトを公開する必要があった．また，ソフトウェア部品を組み込むシステムごとに ID 番号を割り付けるので，ソフトウェア部品はソースで供給しコンパイルする必要があった．

　ソフトウェアグループ管理の導入により，部品を構成するオブジェクトの ID 番号は固定になる．そのため，部品を構成するオブジェクトの増減は，その部品内だけの変更になるので，ソフトウェア部品のバイナリ供給が容易になる．

- 技術開示などの問題によりソース提供ができない場合でも，バイナリ供給が可能になることにより，これまで供給できなかった機能をもつソフトウェア部品が流通する可能性がある．また，バイナリでの供給になると，安価な提供ができるようになる．
- グループを構成するオブジェクトは，他のグループからアクセスできなくなるため，不正なアクセスから保護できる．このことは，障害発生時の原因追求を容易にする．

利用例

以下に「ソフトウェアグループ管理」の利用例を示します．

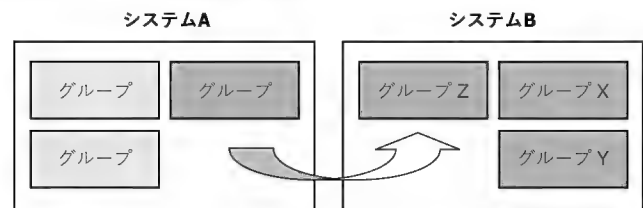
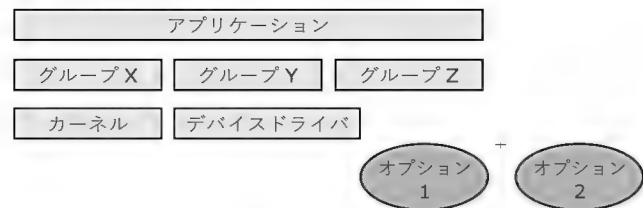
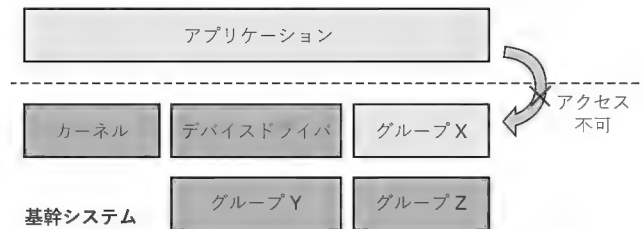
1）ソフトウェア部品化

ソフトウェア部品として，グループ単位に各種のシステムで利用することができます(**図 6**)．

グローバル ID 番号への割り付けが少数であることと，グループを構成するオブジェクトを意識する必要がないことによります．システム A ではグループ 3 として登録していたグループを，システム B ではグループ Z として登録することができます．このようにして，システム構築の生産性を向上させることができます．

2）オプション機能の追加

オプション機能をもつプログラムをソフトウェア部品と扱うことにより，機能追加を容易にします(**図 7**)．

〔図 6〕ソフトウェア部品化の例

〔図 7〕オプション機能の追加例

〔図 8〕基幹システムの保護例

　これも，グループを追加するにあたって，グローバル ID 番号の割り付けが少ないことによります．

3）基幹システムの保護

アプリケーションから基幹システムのオブジェクトに対する不正なアクセスを防止することができます(**図 8**)．

グループ内に管理される ID 番号を割り付けたオブジェクトは，他のグループからのアクセスが防止されます．

また，グループ内に管理された ID 番号を割り付けたオブジェクトに不具合が発生(たとえば，メモリプールのメモリがなくなった)した場合でも，その原因の追求が容易になります．

おわりに

　これまで解説したように，「ソフトウェアグループ管理」は，ソフトウェア部品化により，組み込みシステムの構築を容易にするくふうです．また，保護機能による検証期間の短縮などにより，システム構築の生産性を向上させます．

本機能により，ITRON で問題視されているミドルウェアの流通の活性化に貢献できると思います．

本機能の導入については，info@elmic.co.jp にお問い合わせください．

きんだいち・つとむ　(株)エルミックシステム

組み込みLinuxをとりまく世界

第2回 「組込みLinux評価キット」(ELRK)の概要

渡辺武夫

前回(本誌2003年7月号)は組み込みLinuxのスマートな導入ということで，組み込みOSにおけるLinux導入の考え方から，「組込みLinux評価キット」の前ふりを解説してきたが，今回は，この評価キット“Embedded Linux Reference Kit(ELRK)”について解説する．

「組込みLinux評価キット」(ELRK)とは？

この製品の大きな特徴は，特定の組み込み向けターゲットボードに特化したデバイスドライバなどを含むLinux環境一式が入っていることである．したがってユーザーはとくに開発業務を行わずとも，任意のボードでLinuxを動作させることができる．

Linuxを動作させることを考えたとき，多くの人はPC/ATでWindowsなどを動作させることをイメージするかもしれないが，これが組み込み向けターゲットボードの場合，同様なことを連想するには無理があるといえる．いちばん大きな問題は，Linux起動のために必要なBIOSに相当するものが組み込み向けターゲットボードにないことが原因ではないだろうか．また，Linuxが必要とする各種周辺機器(ペリフェラル)の種類や接続体系がPC/AT機と異なることも理由の一つである．

ELRKは，それらの問題を解決するために，任意の組み込み向けターゲットボード用のBIOS，Linuxカーネルイメージなどの一式をパッケージ化したものとなった．それでは，このELRKに入っているものを一つずつ解説していく．

- BIOS

Linuxユーザーにとって「BIOS」というと，言葉としてはなじみが深いと思うが，それがどんな目的でどんな機能を実現させるためにあるか，いま一つはっきりしていないのではないだろうか．LinuxにとってのBIOSとは，Linux本体を起動させるためのスタータのようなものにしかすぎない．ただし，この機能がないと，Linuxの起動ができないのも事実である．もう少し具体的には，ターゲットボードの電源投入に連動し，各種ハードウェアの初期化を行った後，Linuxイメージを任意のデバイスより取得し，メモリに展開/起動するためのものである．

このような機能(Linuxイメージロード)から，組み込みLinuxではこのBIOSを“ブートローダ”と呼んでいる．したがって，ELRKでもBIOSという言葉は使用せず，ブートローダと記してある．

先にも記したように，PC/AT互換機にLinuxを実装する場合，BIOSが存在するので，これがブートローダとして機能することになるが，組み込み向けターゲットボードの場合，このようなブートローダが実装されているかどうかは決まっていない．もちろん，ボードメーカーが独自に電源投入に連動したプログラムを実装している場合がほとんどだが，それがLinuxをロード/実行させるための機能を満たしているかどうかは定かではないため，必要に応じて，別途ブートローダを実装することが必要となる．

ELRKではブートローダとして，RedBootを採用している．これは，Red Hat社が提供するモニタプログラムであり，次のような機能をもっている．

- 電源投入に連動して起動し，ターゲットボードの初期化を行う

これは組み込みで一般的にいわれるリセット起動を意味する．つまり，パワーオンリセットに連動して動作開始ができ，動作後，ターゲットボードの状態安定(初期化)を行ったり，後に記すLinuxイメージ展開などを実現させるためのメモリの初期化を行っている．

- ターゲットボードに実装されているフラッシュメモリを用い，簡易ファイルシステムを構築し，そこにLinuxカーネルイメージを保存する

RedBoot自身はLinux自体を取り込んでいるわけではなく，どこからかLinuxイメージを読み込まなければならない．試験/評価タイミングであれば，どこから読み込むかはあまり大きな問題とはならないかもしれないが，実際に組み込み製品とした場合，ターゲットボード内部でLinuxイメージを保有していなければ，結果としてLinux動作につながらないことになる．これに対し，PC/AT互換機の場合，ハードディスクや，フロッピディスクなどがあるが，組み込み向けターゲットボードの場合，このような機材が装着されていること自体が稀といえる．したがって，ハードディスクやフロッピーディスクに代わるものとして，本機能が提供されているわけである．

- 任意のペリフェラル(シリアルポート/Ethernetポート/簡易

ファイルシステム)経由でのLinuxカーネルイメージのメモリ展開機能

シリアルポートでの読み込み(ダウンロード)は一般的な作法だが，近年のプログラムの肥大化にともない，シリアル経由でのダウンロードは非常に時間がかかる状況となってきている．それに対応することも含め，RedBootではEthernet経由での高速ダウンロードが提供されるようになってきた．

- 任意のペリフェラル(通常はシリアルポート)をRedBootコンソールとして用いて，テキストベースでの会話によるLinux起動をはじめとした各種機能

*　　　*

このように，RedBoot自体も非常に機能が豊富で，それ単体でさまざまな使い方ができるプログラムなので，機会があったら，ぜひ詳細を読者自身でふれて確認してみてほしい．ただし，一つだけ付け加えさせていただくと，ELRKに同梱されているRedBootは，いままで紹介された機能をすべてもっているわけではない．これは，ELRKの目的にも関係するが，あくまでLinuxを起動させるためのブートローダとして使われているだけなので，ものによっては最小限機能のみとなることもある．ただし，RedBoot自体と，対象となるターゲットボードの差分ソースコードも含まれているので，ユーザーは改造をすることにより，RedBootがもつ全機能を実現させることもできる．

● 組み込みLinuxイメージ/Linux環境

ここが，ELRKでのいちばんの「肝」となる項目である．一般的にLinuxを任意なターゲットボードに実装させる場合，どのような手順を用いて作業を行うだろうか．たいていの場合，非常に似通ったターゲットボード用に構築されたLinux(もしくはLinuxパッチ)をインターネットなどより入手し，それを改造して動作させることになるわけだが，これ自体が非常に労力の必要な作業になる．かりに，そのボードそのものがすでに存在していれば話は別だが，実際にはそう虫のいい話は少ないのではないだろうか．

実際，自分が見つけたターゲットボードにLinuxを実装したい場合，多くは実装手順書があるわけでもなく，さまざまな文献やインターネットを調べてLinuxの構造を理解し，そのうえで，ともかく載せてみようというのが現状だと思う．また，実装が完了した後でも，何をもって完了したのかが疑問となり，自分が本来予定しているその先の業務の足を引っ張ることが多々あるのではないだろうか．

ELRKではまず，「Linuxの動作」という言葉を次の定義に基づいて考えている．

- 所定のシリアルポートをLinuxコンソールとして，各種Linux標準操作はこれに接続した端末経由で行う．また，Linuxが必要なファイルシステムは，ホストコンピュータの所定ディレクトリをEthernet経由でNFSマウントして実現させる

したがって，このLinuxでは非常にシンプルなデバイスのみが動作していることになる．Linuxコンソールを実現させるためのシリアルポートと，NFS機能を実現させるためのEthernetポートのみということである．このように記すと，非常に簡素で何も機能がないように見えるが，前述したように，どのような定義でLinuxを実装したかを明確にし，その先はユーザーにバトンタッチするということが一つの目的となるわけである．

つまり，ユーザーは何も特別なプログラム改造をすることなく，ターゲットボード上でLinuxを動作させる第一歩を踏み出せることになる．また，最初に何からやったらよいのかわからないユーザーを考慮し，すでにビルドしたカーネルイメージを同梱している．結果として，とりあえずLinuxを動かしたいと考えているのであれば，同梱されているカーネルイメージをターゲットボードで動作させれば，Linux起動ができるというわけである．もしも，ユーザーがその先にターゲットボードに実装されている，その他のペリフェラルを動作させたいと考えるのであれば，ドライバ組み込み作業のみに着目すればよいことになる．

次に，使用しているLinuxがMontaVista社の組み込み向けLinux(MontaVista Linux)ということも一つの「肝」といえる．MontaVista Linuxは，組み込み用として次の機能が実装されている．

- プリエンプティブカーネル
- リアルタイムスケジューラ

この両機能は，組み込みオペレーティングシステムとしては非常に有意義な機能といえる．この機能のおかげで，従来のLinuxが組み込み機器に適していないといったイメージを打破することができたといえる．個々の機能はカーネル再構築をするだけで使用する/しないを選ぶことができるので，自分の作成したアプリケーションが個々の機能とどのように連動して動作するのかを確認することもできる．実際のところ個々の機能は，単純なアプリケーションでは効果を示さない機能なので，この機能の効果を見る場合，それなりのアプリケーションが必要となるが，とりあえず，機能を試してみるといった評価のとっかかりができているのもELRKの特徴となっている．具体的な個々の機能の詳細については，今回は誌面の都合上，省略させていただく．

● クロスコンパイラ

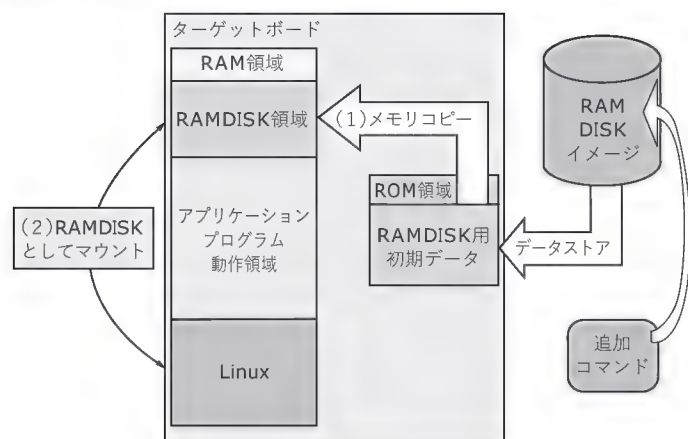
ELRKには，各種ソフトウェアを再構築するためのクロスコンパイラが同梱されている．これ自体は目的から考えるとあって当たり前だが，いざ自分達で環境をそろえるとなると，それもたいへんな作業となるのではないだろうか．実際，クロスコンパイラだけではなくCライブラリ(GLIBC)や，ヘッダファイルなどをそろえるとなると，それだけで一仕事だと思う．

● ターゲット動作コマンド群(ユーザーランドプログラム)

最後に，忘れがちなことなのだが，Linuxを使用する場合，コマンド群がそろっているかどうかも，作業を円滑にするために必須な項目といえる．ただし，このコマンド群の数は，即座にターゲットのメモリ量と連動してしまっているのが現状であ

〔図1〕コマンド追加のイメージ

〔図2〕ELRKの動作体系

〔図3〕カーネルイメージの構築

る．つまり，多くのターゲットボード向けLinuxは良くも悪くも，この部分を，RAMDISKなどオンボードメモリをストレージとした環境で提供されるため，実際のターゲットで使用できる数や，個々のアプリケーションの脱着があまりスムーズとはいえないと考えられる．

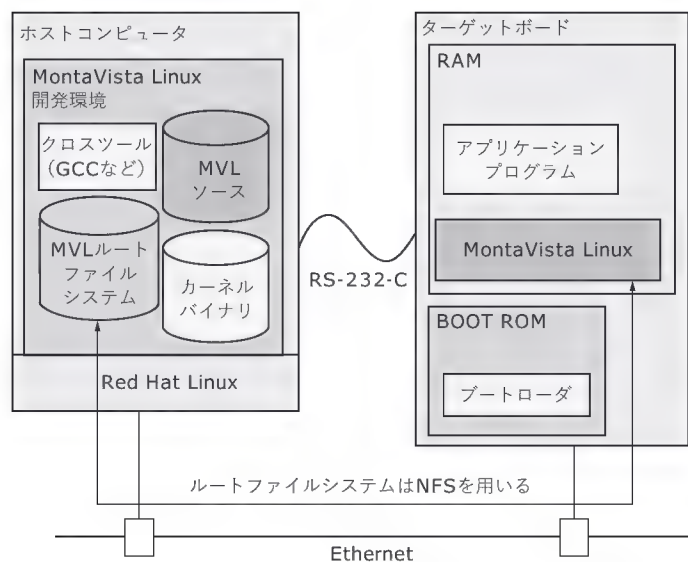
つまり，提供された環境にコマンドが足りない場合や，機能を変更したい場合，そのアプリケーションを追加/変更しなければならないが，その手法がある程度のLinux知識を保有している必要がある，と考えられる．たとえばコマンドを追加したい場合，最初に追加するコマンドの実態である実行イメージを，オリジナルのディスクイメージに追加し，そのディスクイメージをターゲットボードに転送し，その後，Linuxを起動する，といった，人によってはなじみのない言葉が連発する方式となる可能性がある（**図1**）．

ELRKの動作構成

図2にELRKの動作体系を示す．ELRKは前述したように，開発環境と動作環境が異なる“クロス開発環境”を基準としている．Linuxユーザーからは，多少イメージしづらい構成に見えるかと思うので，ここであらためて，その動作を順に解説していく．

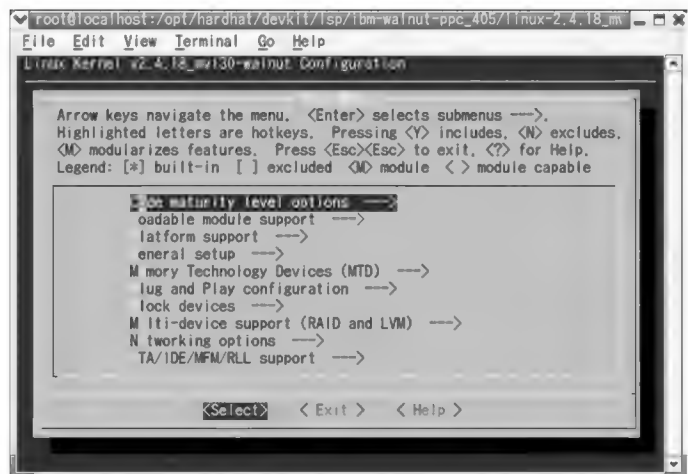
● ステップ1：ブートローダの装着

ELRKでは，対象となるボード用のブートローダが同梱されている（注意：PC/AT互換機が対象ターゲットボードの場合，BIOSがあるので，ELRKにはブートローダは入っていない）．最初に使う人は，ブートローダをターゲットボードの起動メモリ（通常はブートROM，ブートフラッシュROMなどと呼ばれる）に書き込む必要がある．実際，書き込み方法はターゲットボード固有な操作となるが，一般的な方式として，フラッシュROMの場合，ICEやJTAGエミュレータなどを使ったり，ROMの場合ROMライタなどを使うのがほとんどであろう．ただし，ターゲットボードによっては，ボードメーカーがターゲットボードに事前に実装したモニタプログラムでブートローダを書き込めることもある．

● ステップ2：Linuxカーネルイメージの構築

ELRKではすでに構築済みのMontaVista Linuxカーネルイメージを同梱しているので，とりあえずLinuxを起動したいユーザーは，このステップを省略できる．また，逆にある程度Linuxに慣れ親しんだユーザーは，MontaVista Linuxを再構成し，カーネルイメージを再構築できる．この操作は，一般的にLinuxで流通している方式と大差はない．実際の動作例を**図3**に示す．これはmenuconfig機能を用いた場合である．一般的な方式としては，ホストコンピュータ上で次のようなコマンドを用いて環境設定，構築を実現する．

`make menuconfig`：これにより環境設定画面が現れる

`make deps`：上記の操作の反映

make zImage(make zImage.srec):実際のカーネルイメージの作成

● ステップ3:ホストコンピュータの設定

さて,Linuxカーネルイメージが完成したら,即座に動作させたいところだが,その前にホストコンピュータの設定をしなければならない.どのような設定を行うかをキーワードを基準に説明する.

1) DHCP(BOOTP)サーバ[注1]

標準で構築されたLinuxは,自分自身のネットワーク情報(IPアドレス)の決定を,DHCPサーバにゆだねる構成となっている.したがって,ホストコンピュータにDHCPサーバ(もしくはBOOTPサーバ)が必要となる.

2) NFSサーバ[注2]

ターゲットボードで動作するMontaVista Linuxは,Linux動作のためのルートファイルシステムをNFS機能で実現させている.したがって,ホストコンピュータでNFSサーバ機能が動作している必要がある.また,exportするディレクトリは,製品同梱の所定ディレクトリとなる.

● ステップ4:MontaVista Linuxの実行

ELRKでは,MontaVista Linuxのカーネルイメージの実装方法を規定していない.一般的なターゲットボード同梱の組み込みLinuxの場合,Linuxカーネルイメージもターゲットボードのフラッシュ ROMなどに書き込むことを実現しているが,ELRKでは,その目的として,ユーザーがLinuxカーネルを何度も再構築して動作させることを想定し,極力Linuxカーネルイメージをホストコンピュータから逐次,転送することを推薦している.たしかに,この方法だと,ターゲットボード起動ごとにLinuxカーネルを転送しなければならないので,Linuxの起動だけでも手間が増えると考えられるが,先に記したように,ただLinux(MontaVista Linux)を動作させるだけではなく,その機能を評価すると考えると,ユーザーは結局のところ,毎回カーネルの再構築をすることがあり得るので,あながち,無駄な方法とはいえないと思う.

● ステップ5:Linux操作

正常にLinuxが起動すると,Linuxコンソール(ELRKでは特定のシリアルポートで考えている)経由で,一般的なLinux操作とまったく同じ操作方法でMontaVista Linuxの操作ができる.したがってユーザーは,キャラクタ端末(Linuxホストの場合minicom)をシリアルポートに接続しておくだけである.

● ステップ6:アプリケーションの追加/削除・Linux動作環境の変更

ユーザーが独自に作成したアプリケーションプログラムは,ルートファイルシステムとしてマウントしているNFSディレクトリにコピーすれば,即座にターゲットボードでも動作可能なので,あとは,普通のLinuxのプログラム起動と同じようにコマンド感覚で起動するだけである.同様に,NFSマウントされているホスト側ディレクトリには,さまざまなLinux環境設定(俗にいうrc.dなど)があるので,そこをホスト上で変更し,ターゲットボード側のMontaVista Linuxを再起動するだけで,Linux動作環境を変更することができる.

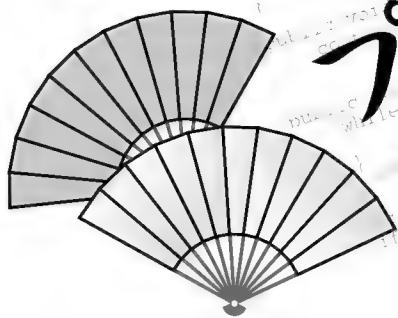
おわりに

近年は,Linuxを組み込み製品のOSとして使うことに興味をもつ方が多いと思う.しかし,Linux自体を組み込み向けに使用する場合,実際に自分が目的とする環境(ターゲットボード)でLinuxを動作させるために,慣れていないとOSを動作させるだけで,非常に大きな時間と手間をかけてしまっている現状もある.本質はLinuxを動作させることではなく,その上でのアプリケーションや,ドライバ開発といった,システム固有の業務が本来の目的であり,Linuxの実装に手間ひまをかけたくないというのが,多くの方の願いであると思っている.

ELRKでは,とりあえず自分が使いたいと思えるボードで,組み込みLinuxとして定評のあるMontaVista Linuxを事前実装しており,これだけで最初の垣根を超えられると信じている.また,何かを評価したい(OSを評価したいなど)という言葉をよく聞くが,どんな目的でどんな内容を基準としてどんな評価を行うかがあいまいになってはいないだろうか.これに対し,非常にシンプルだが,前述した機能とMontaVista Linuxを変更させるのに必要な環境一式が入っているELRKは,その先はユーザーのアイデアしだいになるものの,MontaVista Linuxをさわり,さまざまな試験(評価)を行うための手助けの選択肢の一つとして,検討の俎上にのるのではないかと思う.

わたなべ・たけお (株)イーエルティ

注1:DHCPサーバの存在について――通常,DHCPサーバは同一イントラネット空間に一つしか存在することが認められていない.したがって,可能であれば,ホストコンピュータとターゲットボードは,社内のネットワークとは別にしておくことが望ましい.

注2:NFSサーバのIPアドレスについて―― MontaVista Linuxはその動作のためのNFSサーバのIPアドレスについて,DHCPサーバのIPアドレスを用いることにしている.つまり,IPアドレスを提供してくれたサーバが同時にNFSサーバとなると考えているわけである.

宮坂電人

第6回

詳細と抽象

抽象化以前の問題

これは筆者がある現場で目撃した笑い(?!)話です．もっとも，当事者たちにとっては笑えない話でしたが．

単純な入力ルーチン(**リスト1**)があったのですが，何かの事情で‘A’から‘J’までしか入力できないようフィルタをかけたバージョンも欲しいと要求されました．たいして難しくないので，要求された側は**リスト2**のような関数を提供しました．さらに翌日，今度は‘A’から‘K’までしか入力できないよう制限をかけたバージョンをという要求があり，同じように対応した関数を提供しました．単にLineInputExをコピー＆ペーストして‘J’を‘K’に書き換えるだけの手間ですから．

〔リスト1〕単純な入力ルーチン

```
void LineInput(char *oBuffer,int iBuffSize)
{
    char aKey;
    do{
        aKey = OneKeyInput(); /* 1文字をキー入力 */
        if(aKey == CR){
            *oBuffer = '¥0';
            return;
        }
        *oBuffer++ = aKey;
        --iBuffSize;
    }while(iBuffSize > 1);
    *oBuffer = '¥0';
}
```

〔リスト2〕‘A’から‘J’までしか入力できないルーチン

```
void LineInputEx(char *oBuffer,int iBuffSize)
{
    char aKey;
    do{
        aKey = OneKeyInput(); /* 1文字をキー入力 */
        if(aKey == CR){
            *oBuffer = '¥0';
            return;
        }else if(aKey < 'A' || aKey > 'J'){ /* (追加) */
            Beep(); /* 警告音 */
        }else{
            *oBuffer++ = aKey;
            --iBuffSize;
        }
    }while(iBuffSize > 1);
    *oBuffer = '¥0';
}
```

ところがその翌日，今度は‘A’から‘L’までという要求があり，さすがにこの時点で，「ちょっと待った」となりました．この調子だと明日は‘A’から‘M’までバージョン，あさっては‘A’から‘N’までバージョンを要求するんだろうと，なかば感情的に，要求された側が反論しました．じつをいうと入力制限はその後，いろいろなバリエーションがあり，なかには入力されたものを変形するという日本語入力フロントエンドプロセッサに近い働きのものも出てくるのですが，そのたびにLineInputを作った側がふりまわされたのではたまりません．

結局，入力制限は入力関数を利用する側の問題であって，入力関数側で対処するのはスジ違いという話になりました．ただ，いくらスジ違いでも誰かが入力制限の処理コードを書かねばならないわけで，それをどうするということになり，結果，入力制限のコールバック関数をパラメータとしてつけたバージョンを作りました(**リスト3**)．

これを利用する側は，たとえば‘A’から‘L’までなら，**リスト4**のようにすればいいわけです．ただしこの決定は，双方に感情的なシコリを残す結果になりました．そもそも何が良くなかったのでしょうか．直接の原因は，要求する側に入力制限の機能を抽象化して取り出す考えがなく，最終的な要求だけを相手につきつけたからだと思います．もちろん要求された側もし

〔リスト3〕入力制限フィルタつきの入力ルーチン

```
void LineInputWithFilter(char *oBuffer,int iBuffSize,
int (*iFilterFunc)(char))
{
    int aFiltKey;
    char aKey;
    do{
        aKey = OneKeyInput(); /* 1文字をキー入力 */
        if(aKey == CR){
            *oBuffer = '¥0';
            return;
        }
        aFiltKey = (iFilterFunc != NULL) ? iFilterFunc(aKey) : aKey;
        if(aFiltKey >= 0){
            *oBuffer++ = aFiltKey;
            if(aFiltKey == 0){
                return;
            }
            --iBuffSize;
        }
    }while(iBuffSize > 1 && aFiltKey >= 0);
    *oBuffer = '¥0';
}
```

かり，要求されたことは簡単だからテキパキとコードを書くほうがいいし，多少の追加要求はいつものように「コピー＆ペースト」で対応すればいいとナメていたフシがありました．物事を抽象化して考えるなんて，まどろっこしくて仕事が遅くなるだけだ，と．プログラミングの教科書で「抽象化はたいせつだ」と書かれていても，現場レベルでは，どういう意味なのかを理解していなかったり，そもそも「抽象化？いったい何ですか，それは？」と抽象化自体を知らないというのが実情だったりします．

トップダウンかボトムアップか

プログラムを設計する場合，手当たりしだいに組み立てて一気に全体を完成できるのは，ある程度の規模までです．それを越えると分割して組み立てざるを得ません．その場合，どの“位置”から組み立てるかという観点で，

- トップダウン：上位から組み立てる
- ボトムアップ：下位から組み立てる

の2種類が考えられます．それぞれの特徴として，

- トップダウン：大局は見えやすいが，細部が粗略に扱われやすい．チームプログラムの管理者の受けは良い
- ボトムアップ：細部は見えやすいが，大局が粗略に扱われやすい．個人プログラマの受けは良い

というのもあります．「受けは良い」と書いたのは，別に「ウケ」を狙ったわけではなく，その立場の人にとって“制御しやすい”かどうかを評価したものです．じつはここが困ったところで，ある手法が“制御しやすい”というのは必ずしも「ベスト」を意味しません．それは単に，その手法に慣れ親しんでいるだけなのかもしれません．また，その手法に固執してしまって，その手法では通用しない状況でも無理に同じ手法を適用しようとして失敗する可能性もあります．

トップダウンで攻めた場合

前回の，

- あるファイルから読み取ったデータを圧縮し，別のファイルに書き込む

をトップダウンで設計/製作したとしましょう．与えられた命題が小規模な場合は，すべて1個のメインルーチンの中に納めようと考えます．なぜなら，そうするほうが“制御しやすい”からです．規模が大きいと判断したなら，どれがサブルーチンになりそうかを判断します．すると，

(1) ファイルから読み取るサブルーチン
(2) データを圧縮するサブルーチン
(3) ファイルに書き込むサブルーチン

という3部分に分割できると判断し，それぞれを三つの作業チームに分配するでしょう．ここで，

〔リスト4〕LineInputWithFilterの利用例

```
int filter_A_L(char iKey)
{
    return ('A' <= iKey && iKey <= 'L') ? iKey : -1;
}

void func()
{
    char aBuff[128];
    LineInputWithFilter(aBuff,sizeof(aBuff),filter_A_L);
    ...(略)...
}
```

- さらにファイルだけでなくネットワークや，その他のI/Oにも対応できるようにする

という追加要求が来た場合，せっかく構築した(1)や(3)を改造するよりも(1)や(3)の変形(つまりネットワーク版や他のI/Oアクセス版)を“追加”しようと考えます．なぜなら，そのほうが“制御しやすい”からです．その結果，

(a) 各ルーチン内に“分岐”がたくさん現れる．追加された変形ルーチンの選択を行うため
(b) 分岐のためにパラメータの数が多いルーチンがたくさん現れる

といった弊害が出やすくなります．皮肉なことに，制御しやすくしようとした方策によって，制御しにくい結果がもたらされます．

ボトムアップで攻めた場合

一方ボトムアップではどうでしょうか．この場合，メインルーチンから俯瞰するのではなく，どのようなサブルーチンを用意しようとしたらいいかで設計製作が始まります．すると，

(1) ファイルのアクセスをするルーチン群
(2) データを圧縮するサブルーチン

で攻めていけばいいことがわかります．次に(1)をどうすればいいかでミクロな視点での検討が始まり，ここで，

(1-1) ファイルを開くルーチン
(1-2) ファイルを閉じるルーチン
(1-3) ファイルから1バイトずつ読み取るルーチン
(1-4) ファイルからnバイト読み取るルーチン
……(以下，延々とルーチンが続く)……

というように，ルーチン群の検討が始まります．ある意味，この検討作業はトップダウンのときと似た罠にハマっているのですが，“制御しやすい”ようにするためと思い込んでいるのでハマっていることに気づきません．

検討を続けると，ファイルならではの制約事項や特徴を意識しだし，これを回避したり対応するためのルーチンも考えます．ここで，

- さらにファイルだけでなくネットワークや，その他のI/Oにも対応できるようにする

と追加要求が来た場合，似たようなミクロな視点での検討が始

まり，やはり
(1'-1) ネットワークを開くルーチン
(1'-2) ネットワークを閉じるルーチン
(1'-3) ネットワークから1バイトずつ読み取るルーチン
(1'-4) ネットワークからnバイト読み取るルーチン
……(以下，延々とルーチンが続く)……
とルーチン群の検討が始まりますが，ネットワークにあってファイルにはない"詳細"部分でひっかかります．なぜなら「ネットワークを開く」といっても，どういうプロトコルでアクセスするのかとか，相手をIPアドレスやポート番号で指定するパラメータも余分に必要だとか考え，結局のところ(1'-1)は，厳密には(1-1)と同じ仕様にはなりません．また逆のパターン，ファイルにあってネットワークにはない詳細部分でもひっかかります．すでにファイルにあるルーチンをネットワーク版ではないものにするか，あるいはダミールーチンにするかなど，どんどん"細かい"部分で悩むハメになります．

こうしてせっかくルーチン群を作っても，組み立てるほうはたいへんです．大量の細かい部品を目の前にして，どう組み立てるのか悩みます．苦労して組み立てたルーチンはおもしろいことにトップダウンで攻めたときと同様，
(a) メインルーチン内に"分岐"がたくさん現れる．たくさん用意されたルーチン群の選択を行うため
(b) パラメータの数や種類がさまざまなルーチンがたくさん現れる
といった弊害で頭を悩ませるわけです．ここでも皮肉なことに，制御しやすくしようとした方策によって，制御しにくい結果がもたらされます．

詳細と抽象

ここで，前回出てきたTemplate Methodパターンによる対応を考えてみましょう．これはトップダウンだったのでしょうか，ボトムアップだったのでしょうか．前回は読み取り側を**リスト5**のようにMyAnyReaderというインターフェースで抽象化して考え，書き込み側を**リスト6**のようにMyAnyWriterというインターフェースで抽象化して考えました．

読み書きを"詳細"レベルで検討せずにインターフェースで"抽象"化して考えたのはミクロではなくマクロな視点での検討なので，一見トップダウンのように思えますが，メインルーチンから俯瞰(ふかん)するのではなく，どのようなサブルーチンが必要かという検討があるのでボトムアップとも思えます．しかし，さきほど説明したトップダウンの攻め方やボトムアップの攻め方と決定的に違うのは，さきほどは詳細レベルにこだわった検討作業だったのに対し，Template Methodパターンで対応した場合は詳細レベルではなく抽象レベルから攻めている点です．前回紹介した"依存逆転の原則"すなわち，
(A) 上位レベルのモジュールは下位レベルのモジュールに依存すべきでない．
両方のモジュールは抽象化したものに依存すべきである．
(B) 抽象化したものは詳細なものに依存すべきでない．
詳細なものは抽象化したものに依存すべきである．
でいえば，トップダウン，ボトムアップの攻め方はいずれも"詳細なもの"の追求であり，違いは上位レベルと下位レベルのどちらの位置から攻めるかの話にすぎません．そして詳細なものの追求にとどまる限りは結局，めんどうな状況から逃れようがありません．抽象が詳細に依存し振り回されているため，新たな詳細が現れるたびにリセットされてしまい，再度，抽象を構築し直したり，すでに書き終えた実装をいじり直すハメになるわけです．つまり，こうした依存関係（抽象が詳細に依存する）を逆転して逃れましょうなので"依存逆転の原則"なわけです．

ところで，オブジェクト指向開発が上手な人やチームを観察すると，スタート地点が，
- どのようなクラスが必要になりそうか

という視点なのに気づかされます．「どのようなルーチンが必要になりそうか(ボトムアップ的視点)」でないのに注意してください．また，
- どのようなオブジェクトに分割できるか

という視点もあります．「どのようなルーチンに分割できるか(トップダウン的視点)」でないのに注意してください．ルーチンの準備や分割という視点では，"詳細"から逃れようありません．クラスの準備やオブジェクトの分割という"抽象"レベルに移行すべきです．このあたりの認識が，伝統的な開発手法に染

〔リスト5〕MyAnyReader
```
public interface MyAnyReader {
    public int setUp();              //初期化処理,戻り値が0ならOK,0以外はエラー
    public NSData readAllData();     //読み取ったものをデータオブジェクトにする,エラーならnullを返す
    public void cleanUp();           //後始末処理
}
```

〔リスト6〕MyAnyWriter
```
public interface MyAnyWriter {
    public int setUp();                         //初期化処理,戻り値が0ならOK,0以外はエラー
    public int writeAllData(NSData iData);      //データオブジェクトを書き込む,戻り値が0ならOK, 0以外はエラー
    public void cleanUp();                      //後始末処理
}
```

まっている人ほど，なかなかピンとこないため，オブジェクト指向開発していますとはいうものの，中身は単に昔ながらの手法をそのまま引きずっていているため期待したほど効果が現れないのが実情です．もちろん，昔ながらのルーチンで攻める手法が「慣れ親しんでいる」し「制御しやすい」から，なかなか切り換えられないのでしょうけれど．

抽象化の限界

ただ現実問題として，すべてを完全に抽象に還元できるわけではありません．さきほどの例でいえば，ファイルを開くときとネットワークを開くときの違いで「IPアドレスやポート番号の指定がある」がそれにあたります．上手にネットワークを抽象化できたとしても将来的にIPv6に移行した場合はどうするのかとか，別種のネットワーク規格ならどうするのかとなれば，結局のところ“詳細”からつきつけられた問題は解決できないでしょう．

いくつか解決方法はありますが，とりあえず思いつくところでは，

(A) 詳細の解決をする層を別個に用意する
(B) アダプタを用意する

といったあたりでしょうか．(A)はある意味，詳細との妥協になりますが，あえて目をつぶるというか，清濁あわせ飲む覚悟があれば比較的対応しやすいでしょう．しかしこれも注意しないと，詳細によるリセットで困る事態が起きるかもしれません．

既存の詳細の流用

詳細なものは抽象化したものに依存させるのが得策だとわかっても，すでにできあがっている“詳細”をどうするかという別の問題があります．たとえば**リスト7**のようなネットワークアクセスクラスがすでに用意されていたとします．MyAnyReader，MyAnyWriterの存在を無視しているため，このままでは前回示した圧縮プログラムに流用できません．

いうまでもありませんが，もっとも下手な対応は，このソースを元にMyAnyReader，MyAnyWriterのインターフェースを利用するクラスを新たに作ってしまうことです．それではたいへんな手間がかかりますし，後でSimpleTCPClientにバグが見つかった場合，新たに作ったクラスにもバグが含まれているはずですから，その後始末が二度手間です．そこで，アダプタとなるクラスを新たに作成します(**リスト8**)．

詳細レベルでの差異(つまりファイルにはなくてネットワークにあるIPアドレスやポート番号の指定)はコンストラクタで吸収しています．これを使う例は**リスト9**のようになります．

アダプタ(あるいはラッパ)は，既存のものをなるべく少ない手間で流用できるので便利です．しかし，余計なオーバヘッドがあるので嫌う人もいます．オーバヘッドを除去するために既

〔リスト7〕SimpleTCPClient.java

```
public class SimpleTCPClient {
    //send,recvメソッドの結果
    public class SendRecvResult {
        private int mSize; //読み書きできたサイズ
        private int mErr;  //エラー値(errno)
        public SendRecvResult(int iSize,int iErr){
            mSize = iSize;
            mErr = iErr;
        }
        public int size() {
            return mSize;
        }
        public int err() {
            return mErr;
        }
    }

    //初期化処理
    //戻り値がtrueならOK,falseならエラー
    public boolean initialize() {
        ...(略)...
    }

    //終了化処理
    public void terminate() {
        ...(略)...
    }

    //ホストに接続する処理
    //iHost=ホスト名,iPort=ポート番号
    //戻り値がtrueならOK,falseならエラー
    public boolean connectHost(String iHost,short iPort) {
        ...(略)...
    }

    //ホストに送る処理
    //iMessage = 送りたいデータ
    public SendRecvResult send(byte[] iMessage) {
        ...(略)...
    }

    //ホストから受ける処理
    //oMessage = 受け取ったデータを格納する場所
    public SendRecvResult recv(byte[] iMessage) {
        ...(略)...
    }

    //ソケットを閉じる処理
    public void close() {
        ...(略)...
    }
}
```

存のものを一から作り直す手間を取るか，実行時の多少のオーバヘッドに目をつむるかは状況に応じて判断すべきことなので，どちらが正しいかは一概に決められません．どちらのオーバヘッド(作り直しと検証のオーバヘッドか，実行時のオーバヘッドか)も，状況に応じて無視できるか無視できないかが変わってくるからです．

Bad Design

前回紹介した“依存逆転の原則”を説明しているURL (http://www.objectmentor.com/resources/articles/dip.pdf)に，興味深い一節があったので紹介しておきます．それは，Bad Designを説明している箇所です．いうまでもなくBad Designとは「悪い設計」ですが，designには「構想，着想，たくらみ，意図」という意味も込められています．そのま

〔リスト8〕SimpleNetRW.java

```
import MyAnyReader;
import MyAnyWriter;
import SimpleTCPClient;

public class SimpleNetRW implements MyAnyReader,MyAnyWriter {
    private SimpleTCPClient mClient = null;
    private String mHost;
    private short mPort;
    private boolean mConnected = false;

    public SimpleNetRW(SimpleTCPClient iClient,String iHost,short iPort){
        mClient = iClient;
        mHost = iHost;
        mPort = iPort;
    }

    public int setUp() {                         //初期化処理,戻り値が0ならOK,0以外はエラー
        if(mClient.initialize()){
            if(mClient.connectHost(mHost,mPort)){
                mConnected = true;
                return 0;
            }
        }
        return -1;
    }

    private final int ReadWriteBufferSize = 0x1000;
    private byte[] mRWBuff = new byte[ReadWriteBufferSize];

    public NSData readAllData() {                //読み取ったものをデータオブジェクトにする,エラーならnullを返す
        SimpleTCPClient.SendRecvResult aResult;
        NSMutableData aData = new NSMutableData();
        for(;;){
            aResult = mClient.recv(mRWBuff);
            if(aResult.err() != 0){
                return null;
            }
            if(aResult.size() == 0){
                return aData;
            }
            aData.appendData(new NSData(mRWBuff,0,aResult.size()));
        }
    }

    public int writeAllData(NSData iData) {      //データオブジェクトを書き込む,戻り値が0ならOK,0以外はエラー
        SimpleTCPClient.SendRecvResult aResult = mClient.send(iData.bytes(0,iData.length()));
        return aResult.err();
    }

    public void cleanUp() {                      //後始末処理
        if(mConnected){
            mClient.close();
            mConnected = false;
        }
        mClient.terminate();
    }
}
```

〔リスト9〕SimpleNetRWの使用例

```
    public void test2() {
        SimpleNetRW aNetReader = new SimpleNetRW(new SimpleTCPClient(),"192.168.1.1",(short)20000);
        SimpleNetRW aNetWriter = new SimpleNetRW(new SimpleTCPClient(),"192.168.1.2",(short)20000);
        MyCompress aComp = new MyCompress();
        int aRet = aComp.anyToAny(aNetReader,aNetWriter);
        System.out.println("aRet = " + aRet);
        if(aRet == MyCompress.OK){
            double aCompRate = aComp.compRate();
            System.out.println("aCompRate = " + aCompRate);
        }
    }
```

ま「悪い設計」と訳すと，Bad Designに込められている意味が狭く伝わりそうなので，あえてBad Designのままにしておきましょう．そこにはソフトウェアの設計製作で何かしら悪いことが起きたかどうかを判断するのにTNTWIWHDI criterionで検出するとよいと書かれています．

TNTWIWHDIとは，「That's not the way I would have done it.（それは自分がやったであろう方法ではない）」の頭文字を取ったものです．しかし，このcriterion（判断基準）の弱点

Column

NSDataについて

最近の筆者は，プログラムを検証するのにMac OS XでProject Builderを使っています．前回と今回のサンプルでNSData，NSMutableDataという未知の型が出てきましたが，いずれもMac OS Xの純正APIであるCocoaフレームワークを利用したものです．あえて標準的なクラスを使っていないのは，そのほうがさらっとコードを読み流さず，何が書かれているのか注目するであろうと考えたからです．

NSDataは，簡単にいえばbyte[]のラッパ的クラスです．ただしいったん構築されると，それ以降は中身を変更できなくなります．変更したい場合は，NSDataを継承したNSMutableDataを利用します．

- **iData.bytes（0,iData.length（））**

これは，NSDataにある二つのメソッドを組み合わせています．

- `public byte[] bytes(int start,int length)`：中身に格納しているデータのうちstartで始まる位置からlengthバイト分を返す
- `public int length()`：中身に格納しているデータのサイズをバイト数で返す

- **new NSMutableData（）**

NSMutableDataオブジェクトを作成しています．この状態では中身はないので，appendDataメソッドなどで中身を埋める必要があります．

- **aData.appendData（new NSData（mRWBuff,0,aResult.size（）））；**

これは，コンストラクタとNSMutableDataのメソッドを組み合わせています．

- `public NSData(byte[] bytes,int start,int length)`：bytesのうち，startで始まる位置からlengthバイト分のデータを保持するNSDataオブジェクトを構築する
- `public void appendData(NSData otherData)`：すでに格納している中身の後ろにotherDataの中身を追加する

は，きわめて「主観的」，「感覚的」であるところです．いうまでもなくソフトウェア開発ではどういう人員が割り当てられるかは種々雑多であり，それぞれの人員の主観や感覚がバラバラです．また，時間とともに主観や感覚が変化するので，ある時点で「自分がやったであろう方法」がわかっても，未来の時点でわかる保証はありません．

別のcriteria（criterionの複数形）として，次の現象があるならマズい状況に陥っていることを検出できると書かれています．

（1）Rigidity（融通がきかない）：変更がむずかしい．というのも，なにか変更を加えようとすると他の箇所への影響が多すぎる（原文は「It is hard to change because every change affects too many other parts of the system.」）．

（2）Fragility（脆弱である）：何か変更を加えると，予想もしない箇所が破綻する（原文は「When you make a change, unexpected parts of the system break.」）．

（3）Immobility（よそで使えない）：他のアプリケーションに使い回しがきかない．というのも，そのアプリケーションから取り出せないから（原文は「It is hard to reuse in another application because it cannot be disentangled from the current application.」）．

なにやら本連載の第2回で取り上げた「オブジェクト指向入門」に書いていたことと共通します．いずれにせよ，この三つの状況を反転，すなわち，

（1'）Rigidity（融通がきかない）→ Flexible（融通がきく）
（2'）Fragility（脆弱である）→ Robust（頑丈である）
（3'）Immobility（よそで使えない）→ Reusable（よそで使える）

となればよいのですが，これこそ「言うは易し」というところです．また，RigidityとFragilityが問題であるというのは誰しもが納得するところですが，Immobilityが問題になることが理解できない人がいるかもしれません．というのも，プログラムを作る現場によっては毎回，新規のものを作っているのでReusableなものを作る機会がないとか，Reusableなものを作るのがたいへんだとか，Reusableなもので工程やステップ数が減ると報酬が減るとか（ジョークではなく本当にありうる話），いろいろな理由があって，Reusableなものを作るノウハウが確立していない現場は珍しくないようです．

ただし，上であげた三つの状況は互いに関連性があります．つまり，融通がきかないものは脆弱であったり，よそで使えないことが多いですし，脆弱なものもやはり融通がきかず，よそで使えません．ということは，よそで使えるものは融通がきいて頑丈であるとはいえないでしょうか．実際のところ，よそで使えるようにしようとすると必然的に構造が明確でなければならず，構造が明確になると融通性や頑丈性が高まってくるとも考えられます．

アンチパターン

Bad Designを考える上で無視できないのは，なぜBad Designができてしまうのか．それこそ“Bad Designは自分が望んだであろう事態ではない”はずなのにです．しかし，詳細に分析を重ねると当然のことながら原因があり，恐ろしいことに，いくつかの原因はプログラマ自身やプログラマに仕事を命じている人たち自身が「望んだ事態」であるという皮肉な分析結果が待ち受けているのです．また不幸な原因には，まるでデザインパターンのように，いくつか共通するパターンがあり，それらを“アンチパターン”として分類する人たちもいます．次回は，このアンチパターンなるものを紹介したいと思います．

みやさか・でんと　miyadent@anet.ne.jp

第21回

バックトラッキングによる走査が可能なプログラミング言語 ―― Icon

水野貴明

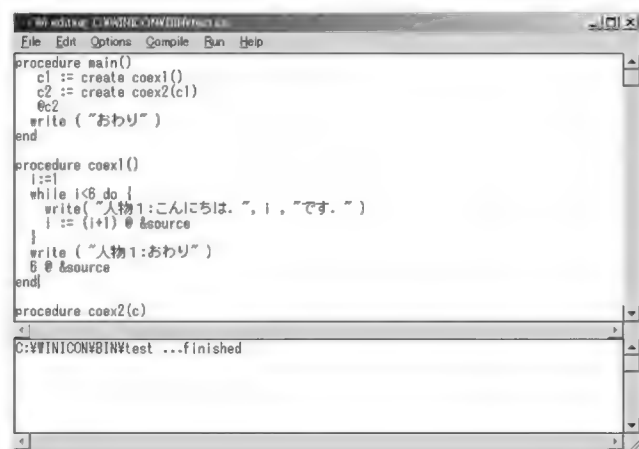
Iconは，アリゾナ大学のコンピュータサイエンス分野のIconプロジェクトで研究，開発が行われており，PDS（Public Domain Software）として公開されているプログラミング言語である．この言語は，1960年代にAT&T（当時）のベル研究所で開発されていたSNOBOLという言語の流れを汲む汎用言語で，それ自身も20年以上の長い歴史をもっている．

この言語はC言語やPASCALなどと似た部分も多いが，いくつかの非常に独特な機能も備えている．今回は，そんなIconの独特な機能を中心に紹介していくことにする．

インストール

Iconは，UNIX版（Mac OS Xを含む）とWindows版が公開されている．ただしWindows版は，UNIX版よりバージョンが若干古くなっている[注1]．

今回は，Windows版をWindows 2000マシンにインストールして検証を行った．Windows版は標準的なインストールファイルになっており，簡単にインストールを行える．標準のインストールパスは「C:\WINICON」である．実行ファイルはその中の「bin」ディレクトリに置かれるので，ここにパスを通しておくとよいだろう．

〔図1〕Iconに付属するGUIツール

DATA

名称：Icon
作者：アリゾナ大学Iconプロジェクト
Webサイト：http://www.cs.arizona.edu/icon/
現在のバージョン：UNIX（9.42），Windows（9.32）
ダウンロードサイズ：1.5Mバイト（UNIX tarファイル）
4.9Mバイト（Windows ZIPファイル）

Iconでは，プログラムファイルをIcon Translatorと呼ばれるツールで実行ファイルに変換し，それを実行する形をとる．UNIX版の場合，Icon Translatorはicont，Windows版の場合はNTICONT.EXEという名前になっている．

たとえば，「test.icn（.icnはIconのスクリプトを表す拡張子）」を実行形式にするには，以下のように単にファイル名を指定すればよい．

```
> NTICON test.icn
```

すると，変換（Translate）が行われ，Windowsであれば「test.exe」という実行ファイルが生成される．「Translate」といっていることからもわかるように，ユーザーが記述したスクリプトはコンパイルされるのではなく，中間コードに変換されているようだ．ただし，実行用のランタイムも実行ファイルに含まれるので，実行ファイルは単体で動作できるようになっている．ファイルサイズも，小さなスクリプトであれば200Kバイト弱程度と，フロッピーディスクにも入るサイズである．

また，Windows版には簡単なGUIの実行環境がついている（**図1**）．これを利用することで，編集から実行までを一つのウィンドウ上で行うことも可能になっている．

Iconのプログラミング仕様

それでは，Iconのプログラム仕様を見ていくことにする．まずは概要を紹介するために，簡単なIconのプログラムのサン

注1：Windowsでも，Cygwinを利用する場合は，UNIX版と同じ最新バージョンを利用することができる．

プルを**リスト1**に示す．Iconの基本的な書式は，CやPASCAL，BASICなどの言語と比較的似ているので，それらの言語を知っていれば読み下すことはそれほど難しくないのではないだろうか．

procedure〜endでくくられるのはプロシージャである．そしてIconのプログラムで最初に実行されるのは，mainプロシージャになる．

代入にはPASCALと同様に「:=」を使う．変数の型は明示的に指定する必要はなく，異なる型に代入した場合には自動的に変換が行われる．

各行の最後にセミコロンなどで明示的に区切りを入れる必要はない．そして，「#」を書くと，その行のそれより後はコメントとみなされる．そして，大文字と小文字は区別される．

ちなみに，writesとwriteは文字を出力する関数である(writeは出力後に改行が行われる)．

〔リスト1〕Iconのプログラムサンプル

```
# Iconプログラムのサンプル
procedure main()
 count := 0                              # 変数への代入
 while( count < 10 ) do {                # count が10になるまでループ
  writes( count, " " )                   # count を表示
  count +:= 1                            # count を1加算
 }
 f:=open("dat.txt","r")                  # ファイルのオープン
 while line := read(f) do {             # ファイルの読み込み
   if find("c",line) then write(line)   # cが含まれていたら表示
 }
 close(f)                                # ファイルクローズ
end
```

成功と失敗

Iconには，「成功(Succeed)」と「失敗(Fail)」という概念がある．Iconにおいて，すべての式は実行された結果，成功/失敗のどちらかの状態になる．そして，制御構文などでは，式が成功したかどうかで条件分岐が行われる．

たとえば**リスト1**に出てきたif文だが，これは以下のような構文となる．

```
if <条件式> then <条件式が「成功」だった場合の処理>
              else <条件式が「失敗」だったときの処理>
```

多くのプログラミング言語では，真(True)と偽(False)という論理値を使って，式が正しいのかそうでないかを表現している．しかし，これはあくまで値である．それに対してIconでは，式はその結果の値とは関係なく，式は必ず成功と失敗のどちらかの状態を取ることになる．

以下のような代入式があったとする．

```
result := param1 > 10
```

「>」という比較演算子は，左辺のほうが右辺より大きければ「成功」となり，右辺の値を結果として持つ．そして，右辺が左辺よりも大きければ「失敗」となり値を持たない，という演算子である．したがってこの場合，param1という変数の中身が10よりも大きければ式は成功となり，resultには10が入る．そして，param1が10以下であった場合は式は「失敗」する．失敗の場合は，代入の処理は行われず，resultにはもとから入っていた値が保持される．

また，次のような式があったとする．

```
result := ( param1 > 10 ) + ( param2 < 20 )
```

式は，一部が失敗した場合は，式全体が失敗したことになる．したがって，param1が10以下であるか，param2が20以上で，どちらかの条件式が失敗した場合，全体が失敗したことになり，resultへの代入処理は行われないのである．両方の条件式が成功した場合は，resultには両者の右辺の和である30が代入される．

同様に，

```
write( param1 > 10 )
```

のような場合は，writeまで含めて「式」であるから，param1が10より大きければ10が出力されるが，10以下であった場合は，式全体が失敗となり，出力は行われない．

この成功と失敗という考え方は，単純な比較などの式の場合は，真と偽を用いたほかの言語の処理と比較してそれほど動作上のメリットはないように思えるかもしれない．しかし，次に紹介するジェネレータとあわせて利用することで，非常にユニークな効果を発揮する．

ジェネレータ

「ジェネレータ(Generator)」は，Iconのもつ重要な特徴の一つである．これは，一言でいうと「複数の値を順番に結果として持つことができる式」ということになる．

たとえば，文字列の中から特定の文字を検索するfindという関数がある．

```
sentence := "abcdeABCDEabcdeABCDEabcde"
write( find("c", sentence) )
```

上記のような処理を実行した場合，最初のcが存在する位置である3が表示される．これは，他の一般的な言語と同じ動作である．しかし，以下のように書くと，状況はちょっと異なる．

```
sentence := "abcdeABCDEabcdeABCDEabcde"
every write( find("c", sentence) )
```

この場合，3，13，23という，変数sentenceに存在するcの位置すべてが表示されるのだ．これは，find関数が，ジェネレータだからである．ジェネレータは，複数の値を順番に生成するもので，この場合findは変数sentenceのなかに存在するcの位置である3，13，23を生成するジェネレータ，ということになる．そして，everyは指定した式にジェネレータが含まれていた場合，そのすべての値を生成するまで式を繰り返し実行してくれる．

ジェネレータには，さまざまなものがある．たとえば，toと

いうキーワードを利用すると，特定の範囲の文字を生成することができる．

```
every write( 1 to 5 )
```

この場合，「1 to 5」という式がジェネレータである．これは，1，2，3，4，5という値を順番に表示する．ほかにも｜を使うと，自由なデータを自由な順番で生成することもできる．

```
every write
        ( "あ" | "か" | "さ" | "た" | "な" )
```

この場合は，「"あ" | "か" | "さ" | "た" | "な"」という部分全体がジェネレータとなる．

続いて，一つの式に複数のジェネレータがあった場合である．

```
every write(( 10 | 20| 30 )+( 1 to 3 ))
```

この場合，式の中には「(10 | 20| 30)」と「(1 to 3)」という二つのジェネレータが存在する．すると，11，12，13，21，22，23，31，32，33という順番で表示が行われることになる．

このような動作が行われるのは，everyが式を「バックトラッキング(back tracking)」というルールにもとづいて実行するからである．

everyは式の評価が終わると，これまで評価した式を逆に走査して，ジェネレータが存在するかをチェックする．そして，ジェネレータが見つかると，そのジェネレータに次の値を生成させ，もう一度，式を評価する．そしてこれを繰り返し，そのジェネレータがもう値を生成しなくなると，さらに式をさかのぼり，ほかにジェネレータがあるかを探す．そして，ジェネレータがあれば，そのジェネレータに新たな値を生成させて式を評価する．

その際，それより後のジェネレータはリセットされるので，再び最初から値を生成する．そして，再び一番最後のジェネレータから値の生成が繰り返し行われる……というように実行されていき，最終的にそれぞれのジェネレータが生成する値のすべての組み合わせが評価されることになる．

このように，式を後からさかのぼっていき，ジェネレータにすべての値を生成させるしくみをバックトラッキングと呼ぶのである．

さらに，式に評価式を追加することで，生成される値から，必要なものだけを選択することができる．

```
every ( i := (1 to 3)) & ( j := (1 to 3))
                          & ( i ~= j ) do {
    writes( i , "+", j , "=", i+j, " ")
    write( i , "-", j , "=", i-j)
  }
```

〔図2〕
goal directed evaluation

```
1+2=3   1-2=-1
1+3=4   1-3=-2
2+1=3   2-1=1
2+3=5   2-3=-1
3+1=4   3-1=2
3+2=5   3-2=1
```

everyは式を評価して，その結果が「成功」だった場合は，do以降の処理を実行する．この処理の実行結果を**図2**に示す．この場合，everyの評価式では，三つの式が「&」で連結された形になっている．一つ目は変数iに1～5の値を入れる式，二つ目は変数jに1～5の値を入れる式，そして三つ目はiとjを比較し，同じだったら失敗となる評価式(「~=」は「=」の逆，つまりiとjが異なる場合に「成功」という意味になる)である．式が評価されるたびに(i ~= j)も同時に評価され，iとjが同じ値だった場合には「失敗」となってdoの後に書かれた処理は実行されない．その結果，iとjが異なる場合にのみdo以降の処理が行われることになる．

このように，ジェネレータと評価式を同時に利用することで，生成された結果から，必要なデータだけを抜き出して利用することができるのである．このように，必要なデータだけを簡単に生成，評価することができるしくみをIconでは「goal directed evaluation」と呼んでいる．

ジェネレータの式のバックトラッキングは，everyを使うのがもっともわかりやすいが，everyを利用しなくても，ジェネレータを含む式は式全体が「失敗」と評価されると自動的にバックトラッキングが行われる．

これを利用すると，たとえば以下のようなことができる．

```
writes(" ",1 to 5) + &fail
```

これは，以下のような結果を示す．

```
1 2 3 4 5
```

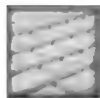
このようなことが起こるのは，「&fail」が常に「失敗」を表すキーワードだからである．Iconには「&」で始まるキーワードがいくつか存在し，それらは状況に応じて特定の値や状態をもつ，システム変数のような働きをする．&failは，その中でも常に「失敗」という状態を保持したキーワードで，これを追加することによって，この式全体が「失敗」と評価されるようになるのである．その結果，バックトラッキングが行われ，ジェネレータはすべての値を生成する．しかも，式全体が失敗であってもwrites自体は「成功」なので文字の出力は行われる．

このような式の失敗によるバックトラッキングは，たとえば以下のような場合に利用できる．

```
sentence := "abcdeABCDEabcdeABCDEabcde"
i :=  find("c", sentence) & ( i > 10 )
```

この場合，iに代入されるのは，3ではなく13である．なぜなら，findは始めに実行されると先頭のcの位置である3が生成され，iに代入されるが，それは続く評価式(i>10)によって「失敗」と評価されてしまう．そして，ジェネレータが含まれたこの式は，失敗によってバックトラッキングが発生する．そして，2番目のcの位置である13が生成されてiに再度代入される．これは評価式(i>10)の条件を満たすので，式は成功し，結果iには13が入った状態となるのである．

ちなみに，ジェネレータは，その式の評価が終わると，初期状態にリセットされる．したがって，たとえば次のように同じジェネレータを繰り返し実行した場合は，毎回3が生成されることになる．

```
sentence := "abcdeABCDEabcdeABCDEabcde"
cnt := 5
while( cnt > 0 & cnt -:= 1 ) do {
    writes(find("c", sentence) , " ")
}
```

プロシージャとジェネレータの作成

Iconでは，main以外のプロシージャを作成することも，もちろんできる．プロシージャを作るには，mainプロシージャと同じようにprocedure～endで囲めばよい．

```
procedure myWrite(x,y)
    write(x,"-",y)
    return "OK"
end
```

プロシージャは，プロシージャの名前とパラメータで呼び出す．プロシージャが，呼び出し位置よりも後ろで定義されていても，問題なく呼び出すことができる．

```
myWrite("ABC","DEF")
```

ここまでは，ほかの言語との違いはほとんどない．しかし，Iconの式はすべて成功か失敗の状態を持ち，プロシージャも式の中で呼び出されるので，プロシージャもどちらかの状態を持つ．

プロシージャでは，処理を呼び出し元に返すためにはreturnを使うことができる．また，処理がendまで行った場合は自動的に呼び出し元に処理が返される．そして，returnを使って明示的にプロシージャを終わらせた場合は「成功」，endまで処理が行って返った場合は「失敗」となる．したがって，以下のような書き方ができる．

```
procedure check(c)
    return if c > 0 then c
end
```

このプロシージャは，渡されたパラメータが0よりも大きいときはその値をそのまま返し，0以下のときには失敗する．

プロシージャを使うと，ジェネレータを作成することもできる．ジェネレータを作成する場合，プロシージャから値を返す際にreturnではなくsuspendを使う．suspendを用いてプロシージャを抜けた場合，そのプロシージャはまだ値を生成できるジェネレータであるとみなされ，バックトラッキングが発生した場合に，suspendの次の行から，処理が再開される．

処理の再開がsuspendの次の行から行われる，というのが大きなポイントで，suspend(一時停止する)という名称のとおり，suspendはプロシージャの実行を一時停止しておいて値を返す，という命令なのである．

これを利用して，次のようにジェネレータを作成することができる．

```
procedure main()
    every write(tofive())
end

procedure tofive()
    i:=1
    repeat {
        if i>5 then break
        suspend i
        i +:= 1
    }
end
```

tofiveは，1から5までを順に生成するジェネレータである．プロシージャ内では，まずiに1が代入され，suspendでその値が返される．そして，バックトラッキングが発生すると，suspendの次の行から実行が再開され，iに1が加算されて，再度suspendでその値が返される．これをiが5を超えるまで繰り返すわけだ．

Co-Expression

Iconの持つ，もう一つの特徴的な機能がCo-Expressionである．これは，特定の処理をパッケージ化するもので，以下のように利用する．

```
c:=create 1 to 3
while writes( " ", @c )
```

createを使うことで，cに「Co-Expression」を格納している．そして，while文で使われている「@c」がCo-Expressionに処理を渡すための命令である．

このプログラムは，「1 to 3」という1から3までの数値を生成するジェネレータをCo-Expressionとしてパッケージ化している．その結果，@cという形で呼び出すたびに，1から3までの値が生成される．Co-Expressionは，一度呼び出すと処理が終わった後も状態が保持され，呼び出すたびに新しい値が生成される．

単にジェネレータを利用するのと，Co-Expressionにして利用する場合の違いは，Co-Expressionにすることで，バックトラッキングが行われなくなる点が挙げられる．したがって，以下のようにeveryを使っても，呼び出しは1回しか行われず，一つの値しか表示されない．

```
c:=create 1 to 3
every writes( " ", @c )
```

その代わり，バックトラッキングが行われない繰り返しを行うwhileを使うことで，すべての値を読み出せる．しかも，バックトラッキングが行われないということを利用すると，次

のようなことができる．

```
c:=create ( 10 | 20 | 30 )
d:=create 1 to 3
while writes( " ", @c + @d )
```

この結果は，「11 22 33」となる．つまり，それぞれのジェネレータが並行的に動作し，同時に新しい値を生成しているのだ．everyを使ってバックトラッキングを行った場合，それぞれのジェネレータの生成する値の組み合わせを「総当たり」させることができた．それが有効な場合も多いが，それぞれのジェネレータに，平行して値を生成させたいこともあるだろう．そのような場合，Co-Expressionが役に立つわけである．

ちなみに，通常のジェネレータはその式でだけ有効で，式を抜けるとリセットされてしまうが，Co-Expressionの場合は，明示的にリセットしない限り，リセットされない．リセットをするには「^c」のように指定する．以下のプログラムは「1231」という出力を行う．

```
c:=create 1 to 3
writes(@c)
writes(@c)
writes(@c)
^c
write(@c)
```

プロシージャをCo-Expressionに指定することで，複数のCo-Expressionで処理を渡しあうことができる．たとえば，**リスト2**のようなプログラムがあったとする．このプログラムの出力結果を**図3**に示す．

〔リスト2〕二つのCo-Expressionで処理を交互に行う

```
procedure main()
   c1 := create coex1()
   c2 := create coex2(c1)
   @c2
  write ( "おわり" )
end

procedure coex1()
  i:=1
  while i<6 do {
    write( "人物1:こんにちは.", i , "です." )
    i := (i+1) @ &source
  }
  write ( "人物1:おわり" )
  6 @ &source
end

procedure coex2(c)
   i:=@c
   while i<6 do {
     write( "人物2:こんにちは.", i , "です." )
     i := (i+1) @ &source
  }
  write ( "人物2:おわり" )
  @ &main
end
```

〔図3〕リスト2の実行結果

```
人物1:こんにちは．1です．
人物2:こんにちは．2です．
人物1:こんにちは．3です．
人物2:こんにちは．4です．
人物1:こんにちは．5です．
人物2:おわり
おわり
```

このプログラムでは，Co-Expressionとして生成されたプロシージャcoex1とcoex2が，お互いに処理を投げあう形で実行されていく．ここで&sourceは自分を呼び出したCo-Expressionを，&mainは始めにCo-Expressionを呼び出したメインの処理を示す．

それぞれのCo-Expressionの処理は，他のCo-Expressionの処理が発生した時点で停止しており，もう一度呼び出されるとそこから再開される．したがって，この処理は**図4**のように進んでいく．

〔図4〕リスト2のプログラムの処理の流れ

このように，Co-Expressionを使うと，複数の処理を交代で行うことができる．これは，たとえば対戦型のゲームにおけるプレーヤの行動のように，複数の処理を順番に行う必要がある場合などに利用できる．

文字列操作

Iconでは，文字列の操作方法にも興味深いものがある．まず，部分文字列へのアクセスは，以下のように大括弧で位置を指定することで簡単にできる．

```
strdata := "abcde"
write( strdata[2] )      # 「b」が表示される
write( strdata[2:4] )    # 「bc」が表示される
write( strdata[-1:-2] ) # 「d」が表示される
```

大括弧で指定するのは文字列中の位置だが，そこで指定される数値は文字そのものを差すのではなく，**図5**のように文字と文字の間を指定するものだ．そのため，[2:4]という指定をした場合，2番目と3番目の文字を指定することになる．[2]のように値を一つだけ指定した場合は，その場所から1文字分を

表す．そして，数値がマイナスの値だった場合は，後ろから数えた位置になる．

Iconでは，部分文字列に文字を代入することも可能である．その場合，もとの長さと代入する長さが違ってもかまわない．

```
strdata := "abcde"
strdata[2:5] := "BBCCDD"
    #  strdataは「aBBCCDDe」となる
```

文字列の長さは，*strdataのようにアスタリスクを前につけることで取得できる．また，!strdataのように前にエクスクラメーションマークをつけることで，先頭から1文字ずつを順番に生成するジェネレータを作ることもできる．

```
strdata := "abcde"
write( *strdata , "文字です" )
    # 「5文字です」と表示される
every writes( !strdata, " " )
    # 「a b c d e」と表示される
```

さらに，Iconでは文字列の走査を行うことができる．これは，特定の文字列の中を「現在の位置（データベースにおけるカーソルのようなもの）」を移動させながら処理をしていく，という文字列走査法である．たとえば，以下のようになる．

```
strdata := "This is a pen."
separator := " "++"."
strdata ? {
    while write(tab(upto(separator))) do
                tab(many(separator))
    if &pos < *separator + 1 then
            write(tab(upto(separator)))
}
```

これは，文字列に含まれる単語を分解して表示する．文字列変数の後に「?」をつけると，その文字列をターゲットとした文字列走査を行うことができる．文字列走査が始まったとき，文字列中の現在位置はいちばん先頭（つまり1）に置かれている．

tabは，指定した位置まで現在位置を移動し，現在位置から移動先の間にある文字列を返す関数，uptoは現在位置よりも後に指定した文字が存在するかを調べ，その位置を返す関数である．そして，separatorはキャラクタセットと呼ばれる変数型で，文字の集合体を表すことができる．これを使うことによりuptoなどで「この中のどれか一文字」という指定の仕方ができる．tabとuptoを使うことで，空白で区切られた単語を一つ，切り出すことができるのだ．

さらにmanyは，現在位置よりも後に指定した以外の文字が存在するかどうかを調べ，その位置を返す関数である．これで空白をスキップして，現在位置を次の単語の先頭まで移動する．

そして，&posは現在位置を表すキーワードである．最後に，もしまだ現在位置が文字列の最後まで来ていなかったら，最後の単語を同様に切り出して表示している．

このように，Iconでは「現在位置」という概念を使って文字列を走査できるので，柔軟な文字列処理を可能にしているのである．

〔図5〕Iconでは文字列の間の位置で文字の範囲を指定する

おわりに

今回は，Iconという言語の，とくにユニークな機能に注目してその仕様を紹介してきたが，いかがだっただろうか．

ちなみに，Iconはそのほかにいくつもの機能を備えている．たとえば，リストや連想配列の機能も備えているし，レコードというCの構造体に似た機能もある．

またIconでは，ライブラリと呼ばれるプログラムが用意されており，それを使用することにより，さまざまに機能を拡張することができる．標準ライブラリの中には，正規表現を実現するものや，GUIの機能を実現するものも存在する．ちなみに，Windows版についてくるGUIの実行環境はIconで作られたもので，ソースもついているのでGUIプログラムを作る際の参考になるだろう．

Iconは非常にさまざまな機能をもっており，応用範囲も非常に広い言語である．とくに，ジェネレータとそれを利用した「goal directed evaluation」のおかげで，たくさんのデータの組み合わせの中から，特定のものだけを抜き出して処理を行う，といったことには，とくに威力を発揮してくれそうだ．

しかも，単体で動く実行ファイルを作成でき，しかもそれほど大きなファイルにはならないので，ちょっとした処理をプログラミングしたり，それをだれか別の人に渡したり，といった場合にも活躍してくれるだろう．

なお，Iconに関する詳しい情報は，IconのWebサイトでも紹介されている「The Icon Handbook」というPDFドキュメントに詳しい．また，その他に，何冊かIconの書籍が出版されており，そのいくつかは，やはりPDFで公開されている．興味のある方は，これらのドキュメントを参考にしてほしい．

1) The Icon Handbook
  http://www.toolsofcomputing.com/IconHandbook/
2) その他の書籍
  http://www.cs.arizona.edu/icon/books.htm

みずの・たかあき

画像実験ソフト「IPキットIII」を使った

画像検査アルゴリズムの検証

石井　均

半導体やプリント基板など電子機器の製造では，あたりまえの技術として画像検査装置が使われています．このような産業向けの画像処理装置の開発は，「画像処理専門」の筆者の会社にとってとても身近なテーマであり，もっとも得意とする仕事の一つです．そこで今回は，われわれの技術の紹介も兼ね，このあたりのノウハウを解説します．

1 開発アプローチ

今回は，「欠陥検査装置」の開発を想定します．具体的には，BGAタイプの電子部品に対してハンダボールのショートを検出します．開発は，次のアプローチに沿って進めます．

①検査の目的の整理→②必要な機能の洗い出し→③機能を実現する処理の検討→④実験用のソフトなどで検証→⑤実用化の検討

実験には，筆者の会社で開発した画像処理ソフト「IPキットIII」を使いました．「IPキットIII」は，これから説明するような「定石」といわれる画像処理を簡単に検証できます．今回は，試用版(3か月間無償使用可能)の「IPキットIII」(**図1**)を本号付属CD-ROM「InterGiga」に収録しました(筆者の会社のWebからも入手できる．http://www.kitech.co.jp/)．興味をもたれた方はぜひ，実験の内容を試してみてください．

〔図1〕IPキットIIIの画面イメージ

2 目的の整理＆処理の検討

前述した開発アプローチにしたがって，目的・機能・および処理を，次のように整理しました．

- **検査目的：** BGAタイプの電子部品の，ハンダボールのショートを検出する
- **必要な機能：**

① 入力画像から光の反射やノイズを軽減する機能
② サンプル(各ボール)を抽出する機能
③ サンプルの特徴を測定する機能

- **機能を実現する処理：**

① 前処理(各種フィルタリング)
② ラベリング
③ 特徴量測定

検査対象の部品には，256個のハンダボールがあります．もしショートしているボールがあれば，②の機能で，ボールの数が少なく撮影されるはずです．また，ショートしている箇所ではボールの面積の外形が変わるはずです．ボールの外形や面積は③の機能で知ることができます．

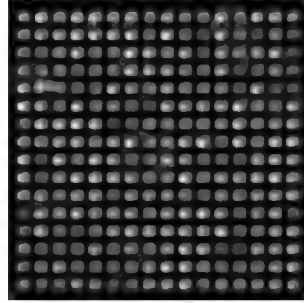
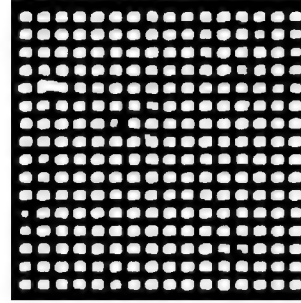
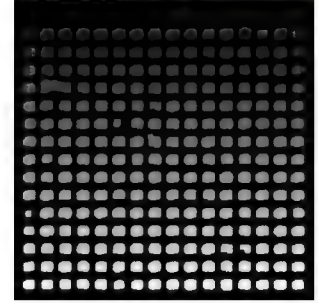
3 画像実験ソフト「IPキットIII」による実験＆検証

① 前処理

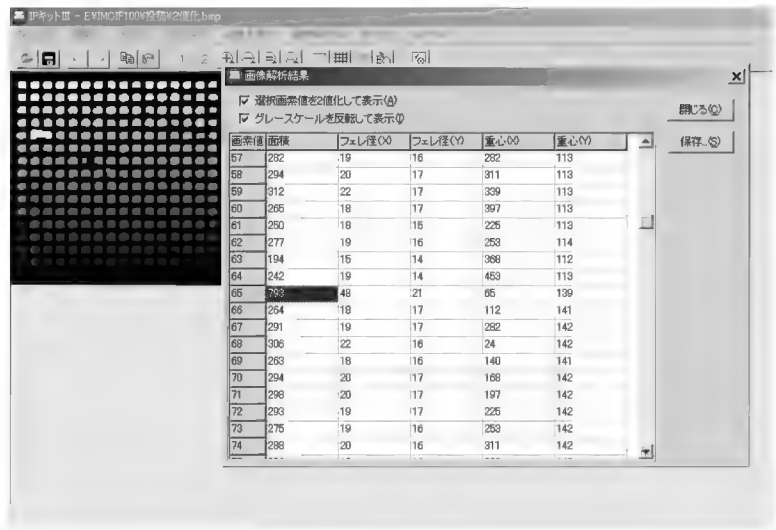
サンプル画像を使い，アプローチの想定を検証してみます．ボールと背景の明るさがまったく異なる画像なので，グレースケールに変換した画像を入力画像とします(IPキットIIIでは「前処理」メニューの「階調・色数」にある「グレースケール画像に変換」を使う)

グレースケールに変換しただけだと，光の反射のためかボールの内側で色ムラができてしまいました(**図2**)．そこで，最大値フィルタ処理を行います(「前処理」メニュー，「グレースケール用フィルタ」の「最大値」を使う)．色ムラを減らし，ボールと

〔図2〕グレースケールに変換した入力画像

〔図3〕最大値フィルタ処理した結果

〔図4〕2値化した画像

〔図5〕ラベリングした結果

〔表1〕「面積」と「フェレ径」，「重心」

画素値	面積	フェレ径（X）	フェレ径（Y）	重心（X）	重心（Y）
1	255	17	17	226	27
2	277	19	16	25	28
3	297	20	17	54	28
:	:	:	:	:	:
64	242	19	14	453	113
65	793	48	21	65	139
66	264	18	17	112	141
:	:	:	:	:	:
253	298	20	16	425	455
254	272	19	16	453	456
255	209	17	14	168	456

〔図6〕特徴量測定

背景の明るさの差を大きくすることができるはずです．所望の結果を得られました（**図3**）．

最後に2値化し（「前処理」メニュー，「階調・色数」の「白黒2値画像に変換」で行う），ボールは白，背景は黒と区別できるようにします（**図4**）．

② ラベリング

ボールと背景とが区別できるようになったところで，ボールとその周りのボールとの区別する処理をします．これには「ラベリング」という手法を使います．ラベリングは，画素値が同じで隣りあう画素に共通の番号（ラベル番号）を割り付ける処理です．「IPキットIII」では，ラベリングをすると（「画像変換」メニュー，「ラベリング」），各ボールを構成するすべての画素が，ラベル番号と同じ画素値になります（**図5**）．ボールは255個しかみつからなかったため，どこかにショートがあるはずです．

③ 特徴量測定

ショートがありそうなので，その場所を調べます．各ボールの特徴量を測定します（「画像解析」メニュー，「特徴量の計測」）．今回の場合，「面積」と「フェレ径」，「重心」が必要となります（**表1**）．ちなみに「フェレ径」とは，ボールの外接矩形の幅と高さを表します．

まずは，表の面積の列を確認すると，画素値（ラベル番号）65のボールの面積が2～3倍ほど大きくなっています（**図6**）．座標は重心から（65，139）のあたりでした．画像を確認してみると，たしかにショートしています．

次に，ショートの方向を調べます．画素値65のフェレ径を見ると，水平方向の値が2～3倍ほど大きくなっています．よって，二つのボールが水平方向につながっていることがわかります．画像もそのとおりになっています．

4 実用化の検討

実験では，2値化のしきい値の決定と特徴量を使った良・不良の判定は，人間が行いました．もしこの処理を実用化するのであれば，装置にこの判断を与えるしくみを検討する必要がありそうです．また，今回の実験は画像が小さく簡単な処理なので，ソフトウェアでも高速でした．ただし場合によっては，コストをかけてもハードウェア化しなければなりません．製品の用途やスペック，価格などから判断する必要があります．

参考文献

1）（株）ケーアイテクノロジー，「検査装置用『画像処理アルゴリズム』の開発入門」，『画像ラボ』，2003年6月号

いしい・ひとし　（株）ケーアイテクノロジー

XScaleプロセッサ徹底活用研究

第3回 USBターゲットプログラミング事例

桑野雅彦

連載第3回の今回は，XScaleに内蔵されているUSBターゲットコントローラを使って，簡単な仕様のUSBターゲット機器を実現してみる．CPUボード上のディップスイッチと7セグメントLEDを，Windowsマシンから制御するまでを解説する．またCPUボード上にプッシュスイッチを実装し，押した瞬間にCPUに割り込みを発生させ，そのタイミングをWindows側に知らせる機能も実装してみる．これらを応用することで，オリジナル仕様のUSB機器を実現することができるだろう． (編集部)

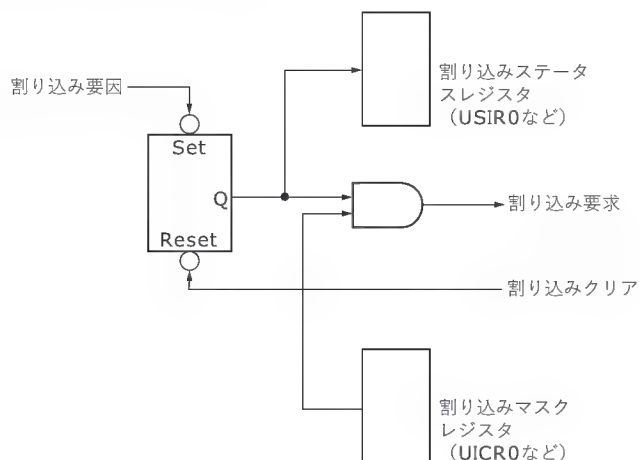
はじめに

CQ RISC評価キットシリーズにIntelのARM系のマイクロコントローラ，XScale(PXA250)を使用した評価キットが追加されました．IntelはXScaleを「Application Porcessor」と呼んでおり，PDAをはじめとする小型の組み込み機器をおもなターゲットとしているのでしょう．

XScale内部には，メモリコントローラ，DMAコントローラ，割り込みコントローラ，LCDコントローラ，AC97コーデックインターフェース，I²CやMMC(マルチメディアカード)インターフェース，FIR(高速赤外線通信)，2チャネルのシリアルポート，リアルタイムクロックと盛りだくさんな内蔵インターフェースに加え，USBデバイスコントローラが内蔵されています．

CQ RISC評価キット/XScaleのCPUボードでは，外部インターフェースとして，シリアルポートやIrDAに加え，USBコネクタも標準で搭載されており，PCなどと接続してUSB周辺機器として利用することができるようになっています．

今回は，このXScaleに内蔵されたUSBインターフェースを利用してみることにしました．

〔図1〕XScaleの割り込み

1 XScaleのUSBデバイスコントローラ

● XScaleの割り込み使用上の注意

XScaleのUSBデバイスコントローラ(UDC)について解説する前に，XScaleの割り込みまわりについて説明します．

XScaleのUSB関係の割り込みは，バスリセットやレジューム，エンドポイントごとの入出力イベントなどがありますが，これらは1本の割り込み要求にまとめられてIR11に入ります．したがって，割り込みコントローラのICMRレジスタなどでは，ビット11がUSB割り込み関係のビットになります．

USB割り込みを使用するうえで注意しなくてはならないのは，XScaleの割り込みがレベルトリガとなっている点です．回路図で書くと**図1**のようなものだと思えばよいでしょう．

つまり，マスクされていない割り込みが発生した場合，それに対応する割り込みを処理して要因ビットをクリアするか，あるいは該当する割り込みをマスクしない限り，いくら割り込み要求レジスタのビットをリセットしてもすぐに再セットされてしまい，延々と割り込みが入り続けてしまうのです．

PC/ATなどでは割り込みコントローラ8259A相当品をエッジモード(正確にはエッジ＆レベル)で使用しているので，割り込みのエントリ部分で，割り込みコントローラのインサービスレジスタの該当ビットをクリアしてから，該当する割り込み処理を行うのが一般的ですが，XScaleでは先に割り込み要因に応じた処理をして，割り込みの発生原因を取り除いてからレジスタのクリアを行わないと正常にクリアできないわけです．

また，使用しない割り込みはきちんとマスクしておかないと，思わぬ割り込みの発生によって割り込み処理ばかりが延々と繰り返され，ハングアップしたような状態に悩まされることになってしまうということにも注意が必要です．

● XScaleのUDC

XScaleのUDCのエンドポイント構成を**表1**に示します．XScaleのエンドポイントは16本あり，それぞれのエンドポイントアドレスやバッファサイズ，種別，方向は固定となっています．

エンドポイントバッファのサイズも固定で，コントロールエ

〔表1〕XScaleのUDCのエンドポイント構成

エンドポイントアドレス	種類	エンドポイントバッファサイズ
0	コントロール(IN/OUT)	16バイト
1	バルクIN	64バイト×2バンク
2	バルクOUT	64バイト×2バンク
3	アイソクロナスIN	256バイト×2バンク
4	アイソクロナスOUT	256バイト×2バンク
5	インタラプトIN	8バイト
6	バルクIN	64バイト×2バンク
7	バルクOUT	64バイト×2バンク
8	アイソクロナスIN	256バイト×2バンク
9	アイソクロナスOUT	256バイト×2バンク
10	インタラプトIN	8バイト
11	バルクIN	64バイト×2バンク
12	バルクOUT	64バイト×2バンク
13	アイソクロナスIN	256バイト×2バンク
14	アイソクロナスOUT	256バイト×2バンク
15	インタラプトIN	8バイト

ンドポイントが16バイト，インタラプト伝送用のエンドポイント(EP5/10/15)が8バイトのシングルバッファで，バルクIN/OUT，アイソクロナスIN/OUTエンドポイントはそれぞれ64バイト，256バイトのダブルバッファになっています．

● USB関連レジスタの種類

XScaleのUSB関連レジスタの一覧を**表2**に示します．かなり数が多いように見えるのは，エンドポイントごとにもっているレジスタが多いためです．これらを整理すると，次のような3種類，計七つのグループに分類することができます．

▶UDC全体の動作に関係するレジスタ

(1) UDCCR(UDCコントロールレジスタ)
　UDCのイネーブルやリセットやサスペンド/レジューム関係

(2) UICR0/UICR1(UDC割り込みコントロールレジスタ)
　エンドポイントごとの割り込みの発生許可/禁止の制御

(3) USIR0/USIR1(UDCステータス/割り込みレジスタ)
　各エンドポイントごとの割り込み発生ステータスリード/クリア

(4) UFNLR/UFNHR(UDCフレームナンバレジスタ下位/上位)

▶OUT方向(ホストからターゲット)のエンドポイント用のレジスタ

(5) UBCRx(xは2/4/7/9/12/14)(UDCバイトカウントレジスタ)

▶エンドポイントごとに用意されているレジスタ

(6) UDCCSx(xは0～15)(UDCコントロール/ステータスレジスタ)

(7) UDDRx(xは0～15)(UDCエンドポイントデータレジスタ)

このうちUDCCSxはエンドポイントの種別によって若干違いはありますが，UDCCS0が若干特殊なほかは方向(INかOUTか)による違いのみ，というものが大半で，中身は同じような構成になっています．

〔表2〕XScaleのUSB関連レジスタ

アドレス	レジスタ名	名称
0x4060_0000	UDCCR	UDCコントロールレジスタ
0x4060_0010	UDCCS0	UDCエンドポイント0(コントロール)コントロール/ステータスレジスタ
0x4060_0014	UDCCS1	UDCエンドポイント1(バルクIN)コントロール/ステータスレジスタ
0x4060_0018	UDCCS2	UDCエンドポイント2(バルクOUT)コントロール/ステータスレジスタ
0x4060_001C	UDCCS3	UDCエンドポイント3(アイソクロナスIN)コントロール/ステータスレジスタ
0x4060_0020	UDCCS4	UDCエンドポイント4(アイソクロナスOUT)コントロール/ステータスレジスタ
0x4060_0024	UDCCS5	UDCエンドポイント5(インタラプトIN)コントロール/ステータスレジスタ
0x4060_0028	UDCCS6	UDCエンドポイント6(バルクIN)コントロール/ステータスレジスタ
0x4060_002C	UDCCS7	UDCエンドポイント7(バルクOUT)コントロール/ステータスレジスタ
0x4060_0030	UDCCS8	UDCエンドポイント8(アイソクロナスIN)コントロール/ステータスレジスタ
0x4060_0034	UDCCS9	UDCエンドポイント9(アイソクロナスOUT)コントロール/ステータスレジスタ
0x4060_0038	UDCCS10	UDCエンドポイント10(インタラプトIN)コントロール/ステータスレジスタ
0x4060_003C	UDCCS11	UDCエンドポイント11(バルクIN)コントロール/ステータスレジスタ
0x4060_0040	UDCCS12	UDCエンドポイント12(バルクOUT)コントロール/ステータスレジスタ
0x4060_0044	UDCCS13	UDCエンドポイント13(アイソクロナスIN)コントロール/ステータスレジスタ
0x4060_0048	UDCCS14	UDCエンドポイント14(アイソクロナスOUT)コントロール/ステータスレジスタ
0x4060_004C	UDCCS15	UDCエンドポイント15(インタラプトIN)コントロール/ステータスレジスタ
0x4060_0050	UICR0	UDC割り込みコントロールレジスタ0
0x4060_0054	UICR1	UDC割り込みコントロールレジスタ1
0x4060_0058	USIR0	UDC割り込みステータス割り込みレジスタ0
0x4060_005C	USIR1	UDC割り込みステータス割り込みレジスタ1
0x4060_0060	UFNHR	UDCフレームナンバレジスタ(上位)
0x4060_0064	UFNLR	UDCフレームナンバレジスタ(下位)
0x4060_0068	UBCR2	UDCバイトカウントレジスタ2
0x4060_006C	UBCR4	UDCバイトカウントレジスタ4
0x4060_0070	UBCR7	UDCバイトカウントレジスタ7
0x4060_0074	UBCR9	UDCバイトカウントレジスタ9
0x4060_0078	UBCR12	UDCバイトカウントレジスタ12
0x4060_007C	UBCR14	UDCバイトカウントレジスタ14
0x4060_0080	UDDR0	UDCエンドポイント0データレジスタ
0x4060_0100	UDDR1	UDCエンドポイント1データレジスタ
0x4060_0180	UDDR2	UDCエンドポイント2データレジスタ
0x4060_0200	UDDR3	UDCエンドポイント3データレジスタ
0x4060_0400	UDDR4	UDCエンドポイント4データレジスタ
0x4060_00A0	UDDR5	UDCエンドポイント5データレジスタ
0x4060_0800	UDDR6	UDCエンドポイント6データレジスタ
0x4060_0680	UDDR7	UDCエンドポイント7データレジスタ
0x4060_0700	UDDR8	UDCエンドポイント8データレジスタ
0x4060_0900	UDDR9	UDCエンドポイント9データレジスタ
0x4060_00C0	UDDR10	UDCエンドポイント10データレジスタ
0x4060_0B00	UDDR11	UDCエンドポイント11データレジスタ
0x4060_0B80	UDDR12	UDCエンドポイント12データレジスタ
0x4060_0C00	UDDR13	UDCエンドポイント13データレジスタ
0x4060_0E00	UDDR14	UDCエンドポイント14データレジスタ
0x4060_00E0	UDDR15	UDCエンドポイント15データレジスタ

● USB関連レジスタ

次に，グループ分けされたそれぞれのレジスタについてみていくことにしましょう．

▶ **UDCCR（UDCコントロールレジスタ）**

USBのバスリセット割り込みや，サスペンド/レジュームを制御するレジスタです（**図2**）．今回のサンプルではリセットやレジューム割り込みは使用しないので，REM，SRMともマスク状態（'1'にする）にしています．

実際にバスリセットやサスペンド/レジューム割り込み条件が満たされた場合，RSTIR，SUSIR，RESIRが'1'になり，REMやSRMビットでマスクされていなければCPUに割り込みがかかります．また，CPU側からこれらのビットに'1'を書くと該当するビットがクリアされます．

RSTIR，SUSIR，RESIRはREMやSRMで割り込みがマスクされていても条件が満たされればセットされます．ほかの割り込みステータスレジスタも同様に割り込みをマスクしていてもステータスだけはセットされるので，割り込みをいっさい使わずにステータスポーリングだけでUSBインターフェースの処理を行うことも可能です．

今回は，リセットやレジューム割り込みは使用しないのでREM，SRMともマスク状態（'1'にする）にしています．

UDEはUDC動作のイネーブル/ディセーブルを制御します．フルスピードデバイスの場合，デバイスのコネクト/ディスコネクトはD+ラインのプルアップ抵抗の有無で判定されますが，XScaleの場合，このプルアップ抵抗は内蔵されていないので，UDEを操作してもそれだけではホストからUSB機器のコネクト/ディスコネクトとしては認識されないということに注意が必要です．

通常は汎用I/Oポートを使ってプルアップ制御を行うのですが，評価ボードではD+につないだ抵抗が直接電源に接続されているため，プログラムをダウンロードしUDEをセットしてUSBファンクションをイネーブルにした後でUSBケーブルを接続しなければなりません．これを守らないと，ホストがUSB機器がつながったと思ってデバイスリクエストを出しても無応答となるため，「不明なデバイス」という扱いになってしまいます．

▶ **UICR0/UICR1（UDC割り込みコントロールレジスタ）**

このレジスタはUSBの各エンドポイントごとの割り込みの発生許可/禁止を制御します（**図3**）．現状，UDCのレジスタはすべて下位8ビットのみ有効となっているため，16個のエンドポイントを二つのレジスタでコントロールする形になっています．

UICR0のビット0がエンドポイント0（コントロールエンドポイント），UICR1のビット7がエンドポイント15（インタラプトINエンドポイント）に対応しており，'1'で割り込みが禁止，'0'で許可になります．

▶ **USIR0/USIR1（UDCステータス/割り込みレジスタ）**

各USBエンドポイントごとに割り込み要求が発生しているかどうかを示すレジスタです（**図4**）．'1'で割り込み要求発生を示し，CPUが'1'を書き込むことでクリアされます．UICR0/UICR1でマスクされたエンドポイントからの割り込みは発生しませんが，割り込み要因となる条件が成立すればUSIR0/USIR1のビットはセットされることに注意が必要です．

▶ **UFNLR/UFNHR（UDCフレームナンバレジスタ下位/上位）**

SOF割り込みの禁止/許可の制御やフレーム番号，アイソクロナスOUT（ホストからXScale側）伝送エラーが起きたかどうかを示します（**図5**）．

今回はアイソクロナス伝送は行いませんし，フレーム番号情報やSOF割り込みも不要なので，これらのレジスタは使用していません．

▶ **UBCRx（xは2/4/7/9/12/14）（UDCバイトカウントレジスタ）**

OUT方向のエンドポイントにホストから送られてきたデータのバイト数を示すレジスタで（**図6**），下位8ビットのみが有効です．バッファに残っているデータ数はこのレジスタの値＋1バイトになります（0xFFなら256バイト）．今回のサンプルでは，バッファの読み出しはUDCCSxのRNE（Receive FIFO not empty）を見ながら行っているため，バイトカウント

〔図2〕UDCCRレジスタのフォーマット

アドレス：0x4060_0000

ビット31～8	7	6	5	4	3	2	1	0
（予約）	REM	RSTIR	SRM	SUSIR	RESIR	RSM	UDA	UDE

REM	USBリセット割り込みマスク	1：USBバスリセット割り込み禁止
RSTIR	USBリセット割り込み要求	1：USBバスリセット割り込み発生（1を書くとクリア）
SRM	サスペンド/レジューム割り込みマスク	1：サスペンド/レジューム割り込み禁止
SUSIR	サスペンド割り込み要求	1：サスペンド割り込み発生（1を書くとクリア）
RESIR	レジューム割り込み要求	1：レジューム割り込み発生（1を書くとクリア）
RSM	デバイスレジューム	1：サスペンド状態から強制復帰させる
UDA	UDC（USBデバイスコントローラ）アクティブ	1：UDCはUSBリセット状態ではない 0：UDCはUSBリセットを受けている（Read Only）
UDE	UDCイネーブル	1：USB動作イネーブル

〔図3〕UICR0/UICR1レジスタのフォーマット

アドレス：0x4060_0050

ビット31～8	7	6	5	4	3	2	1	0
（予約）	IM7	IM6	IM5	IM4	IM3	IM2	IM1	IM0

（a）UDC割り込みコントロールレジスタ0（UICR0）

アドレス：0x4060_0054

ビット31～8	7	6	5	4	3	2	1	0
（予約）	IM15	IM14	IM13	IM12	IM11	IM10	IM9	IM8

（b）UDC割り込みコントロールレジスタ1（UICR1）

IM15	EP15割り込みマスク	1：割り込み発生禁止　0：割り込み発生許可
～	～	～
IM0	EP0割り込みマスク	1：割り込み発生禁止　0：割り込み発生許可

レジスタは使用していません．

▶UDCCSx（xは0～15）（UDCコントロール/ステータスレジスタ）

UDCCSxは，各エンドポイントごとの制御やステータスを読み出すためのレジスタです（**図7**）．UDCCS0はコントロール伝送用として少々特殊な配置になっていますが，その他のUDCCSxはIN方向とOUT方向による違いがあることと，アイソクロナスエンドポイントの場合，STALLハンドシェイクがないため，STALL関係のビットが削除されている程度で，ほぼ同じものになっています．

▶UDDRx（xは0～15）（UDCエンドポイントデータレジスタ）

送受信データポートです（**図8**）．エンドポイントバッファとの間のデータの入出力は1バイトずつ行われるので，下位8ビットのみが有効です．

2 UDCサンプルファームウェア仕様

● USBターゲットの仕様概要

今回のサンプルでは，評価ボードのLED表示とディップスイッチの読み込み，およびボードで発生させた割り込みの取得を行うことにしました．ベンダリクエストのほか，インタラプトIN，バルクIN，バルクOUTの各エンドポイントを利用したデータ伝送を実装しました．

評価ボード上の押しボタンスイッチは，前回の解説からデバッガとの接続にシリアルを選択している場合はGP0が使えそうですが，GP0はUSBの電源検出用に使われているので，デバッガとの接続にUSBを使わなくても，USBそのものを使うと使えないことになります．よって今回はGP57（JEXT2コネクタのC24番ピン）にスイッチを増設して，これが押されると割り込みが発生するようにしました．プルアップ抵抗の電源は3.3Vの電源に接続します．わかりやすいところでは，コンデンサC23（部品未実装）の＋側の端子から取り出すとよいでしょう．GNDはJEXT2のC32番に配線されています．スイッチの配線は**図9**に示すようにごく単純なものです．もう少しまじめに（?）やるならば，CRでフィルタを作り74HC14などのシュミットゲートで受けるなどしてチャタリングを防止すべきですが，今回は簡単に済ませました．

スイッチが押されると割り込みが発生するので，この情報をインタラプトINエンドポイントからホストに送られるデータに付加しておきます．ホスト側ではこのステータスを見て，USBターゲット上で割り込みが発生したことを検出し，割り込み発生を示すダイアログボックスを開くようにしました．

● ホストとの間のデータフォーマット

▶ベンダリクエスト

エンドポイント0（EP0）によるベンダリクエストとしては，エンドポイントバッファのデータをフラッシュしてディップスイッチの現在値を読み出すための要求を作りました（**表3**，p173）．

〔図4〕USIR0/USIR1レジスタのフォーマット

アドレス：0x4060_0058

ビット31～8	7	6	5	4	3	2	1	0
（予約）	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0

（a）UDCステータス/割り込みレジスタ0（USIR0）

アドレス：0x4060_005C

ビット31～8	7	6	5	4	3	2	1	0
（予約）	IR15	IR14	IR13	IR12	IR11	IR10	IR9	IR8

（b）UDCステータス/割り込みレジスタ1（USIR1）

IR15	EP15割り込み要求	1：割り込みサービス要求発生中
～	～	～
IR0	EP0割り込み要求	1：割り込みサービス要求発生中

〔図5〕UFNLR/UFNHRレジスタのフォーマット

アドレス：0x4060_0064

ビット31～8	7	6	5	4	3	2	1	0
（予約）	FNLSB							

（a）UDCフレームナンバ下位レジスタ（UFNLR）

アドレス：0x4060_0060

ビット31～8	7	6	5	4	3	2	1	0
（予約）	SIR	SIM	IPE14	IPE9	IPE4	FNMSB		

（b）UDCフレームナンバ上位レジスタ（UFNHR）

SIR	SOF割り込み要求	1：SOFが受信された
SIM	SOF割り込みマスク	1：SOF割り込み禁止 0：SOF割り込み許可
IPE14	エンドポイント14アイソクロナスパケットエラー	1：EP14に入ってるデータは壊れている
IPE9	エンドポイント9アイソクロナスパケットエラー	1：EP9に入ってるデータは壊れている
IPE4	エンドポイント4アイソクロナスパケットエラー	1：EP4に入ってるデータは壊れている
FNMSB	フレームナンバ（上位）	フレーム番号の上位3ビット（ビット8～ビット11）
FNLSB	フレームナンバ（下位）	フレーム番号の下位8ビット（ビット7～ビット0）

〔図6〕UBCRxレジスタのフォーマット

アドレス：0x4060_0010

ビット31～8	7	6	5	4	3	2	1	0
（予約）	BC[7：0]							

BC[7：0]	バイトカウント（Read Only）	FIFO内部に入っているデータはBC[7：0]＋1バイト

ここでは，このコマンドをGET_CURRENT_VALUEという名称にします．bRequest＝0x00がこのリクエストとなり，wValueの下位が0x00のときインタラプトINエンドポイントを，0x01のときバルクINエンドポイントに現在入っている中身を破棄し，新しいデータを詰め直します．

注意が必要なのは，ベンダリクエスト発行のあとすぐにエンドポイントをリードされると，ファームウェアによるエンドポイントのフラッシュが間にあわず，フラッシュ前に入っていたデータを読み出されてしまう可能性があるということです．こ

〔図7〕**UDCCSx レジスタのフォーマット**

アドレス：0x4060_0010

ビット31～8	7	6	5	4	3	2	1	0
(予約)	SA	RNE	FST	SST	DRWF	FTF	IPR	OPR

SA	SETUP アクティブ	1：セットアップパケット到達(1を書くとクリア)
RNE	受信 FIFO ノットエンプティ	1：受信 FIFO は空ではない 0：受信 FIFO は空
FST	強制ストール	1：STALL ハンドシェークを行わせる
SST	ストール送信完了	1：STALL ハンドシェーク完了
DRWF	リモートウェイクアップ機能	1：リモートウェイクアップ機能が SET_FEATURE コマンドでイネーブルされた(Read Only)
FTF	送信 FIFO フラッシュ	1：送信 FIFO をフラッシュ(クリア)する
IPR	IN パケットレディ	1：送信 FIFO にデータが入っている(書込専用)
OPR	OUT パケットレディ	1：OUT パケットが FIFO に入っている(1を書くとクリア)

(a) UDC コントロール/ステータスレジスタ0(UDCCS0)

ビット31～8	7	6	5	4	3	2	1	0
(予約)	TSP	(予約)	FST(*1)	SST(*1)	TUR	FTF	TPC	TFS

TSP	ショートパケット送信	1：FIFO サイズ未満のデータを送信する
FST	強制ストール	1：STALL ハンドシェークを行わせる
SST	ストール送信完了	1：STALL ハンドシェーク完了
TUR	送信 FIFO アンダーラン	1：送信 FIFO でアンダーランが起きた(1を書くとクリア)：このビットは割り込みを発生しない
FTF	送信 FIFO フラッシュ	1：送信 FIFO をフラッシュ(クリア)する
TPC	パケット送信完了	1：パケットが送信完了した(1を書くとクリア)
TFS	送信 FIFO サービス	1：送信 FIFO が空いている(新規データセット可能)

＊1：UDCCSP3/8/13(アイソクロナス IN エンドポイント)には存在しない

(b) UDC コントロール/ステータスレジスタ 1/3/5/6/8/10/11/13/15 (UDCCS1/3/5/6/8/10/11/13/15)

ビット31～8	7	6	5	4	3	2	1	0
(予約)	RSP	RNE	FST(*1)	SST(*1)	DME	(予約)	RPC	RFS

RSP	ショートパケット受信	1：FIFO サイズ未満のデータパケット，あるいは Zero-Length パケットを受信した
RNE	受信 FIFO ノットエンプティ	1：受信 FIFO は空ではない 0：受信 FIFO は空
FST	強制ストール	1：STALL ハンドシェイクを行わせる
SST	ストール送信完了	1：STALL ハンドシェイク完了
DME	DMA イネーブル	1：受信バッファに32バイト未満のデータがあり EOP がきたときに割り込み 0：EOP で割込
RPC	パケット受信完了	1：1パケット分のデータを受信した(エラー/ステータスビットも有効)
RFS	受信 FIFO サービス	1：受信 FIFO に1パケット以上のデータがある

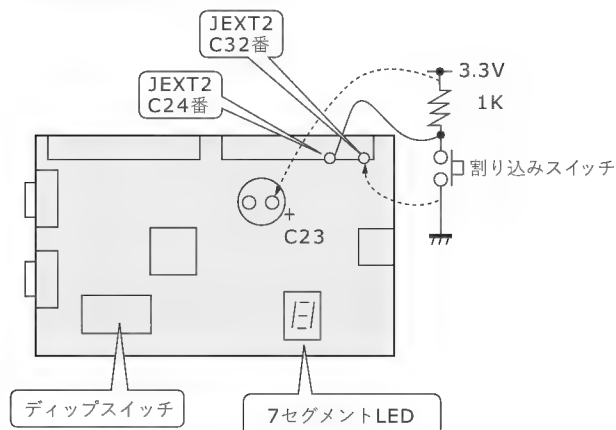
＊1：UDCCSP3/8/13(アイソクロナス IN エンドポイント)には存在しない

(c) UDC コントロール/ステータスレジスタ 2/4/7/9/12/14 (UDCCS2/4/7/9/12/14)

〔図8〕**UDDR レジスタのフォーマット**

ビット31～8	7	6	5	4	3	2	1	0
(予約)	DATA							

DATA	データ	IN エンドポイント：ホストに送るデータ，OUT エンドポイント：ホストから送られてきたデータ

〔図9〕**割り込みスイッチの接続**

の対策のため，データ IN ステージをともなうベンダリクエストとして実装して，動作の同期をとるようにしました．データ IN ステージで読む値自体にはとくに意味がないので，ローカルに展開したディスクリプタテーブルの先頭1バイトを送るだけにしています．

▶バルク OUT データ

バルク OUT(EP2)は常に待ち受け状態に入っており，到達したデータはすべて LED ポートに出力するようにしました．今回のアプリケーションでは，出力パターンを決めてボタンを押すと1バイトのデータが送られて LED 表示に反映されるようにしています．

▶バルク IN データ

バルク IN(EP1)は EP0 で GET_CURRENT_VALUE(BULK)が行われたときにフラッシュし，新しいデータを詰め直します．これによって，GET_CURRENT_VALUE(BULK)のあと，バルク IN エンドポイントのリードをしないまま，再度 GET_CURRENT_VALUE(BULK)が行われても2回目の要求発生時のデータがホストに返されるようになります．

バルク IN で送られるデータは64バイトです．先頭バイトは単純インクリメントで，2バイト目にスイッチ情報を入れてみました．3バイト目以降はとくに意味をもたせてはいません．

▶インタラプト IN データ

インタラプト IN では，8バイトのデータを返すことにしました．データフォーマットを**表4**に示します．先頭の1バイトが割り込みスイッチステータスです．割り込みスイッチによる割り込みが発生すると，割り込み発生ステータスフラグをセット

〔表3〕ベンダリクエスト

バイト位置	フィールド名	データ	備　考
＋0	bmRequestType	0x41	
＋1	bRequest	0x00	
＋2	wValue（L）	0x00 ：インタラプトINデータを更新 0x01 ：バルクINデータを更新	
＋3	wValue（H）	未使用	
＋4	wIndex（L）	未使用	
＋5	wIndex（H）	未使用	
＋6	wLength（L）	0x01	データINステージで1バイト読むことで同期する
＋7	wLength（H）	0x00	

しておいて，インタラプトINでデータを送るときにこのステータスをチェックし，割り込みがあったときには0x01，なかったときは0x00にしています．

2バイト目以降はスイッチのステータスで，2バイト目が最新データです．2バイト目以降は過去のデータで，3バイト目が前回値，4バイト目の更にその前となります．当初どのビットが変化したのかを表示しようと思ったときの名残りで，今回のサンプルアプリケーションでは1バイト目と2バイト目だけ使用しています．

〔表4〕インタラプトINデータのフォーマット

バイト位置	内　容	備　考
＋0	割り込みスイッチステータス	1：割り込み発生 0：割り込み発生していない
＋1	DIPスイッチステータス（最新）	1：該当するスイッチがON 0：OFF
＋2	DIPスイッチステータス（1回前）	
＋3	DIPスイッチステータス（2回前）	
＋4	DIPスイッチステータス（3回前）	
＋5	DIPスイッチステータス（4回前）	
＋6	DIPスイッチステータス（5回前）	
＋7	DIPスイッチステータス（6回前）	
＋8	DIPスイッチステータス（7回前）	

3 ファームウェアの設計

● ベンダ/プロダクトID

USB機器として認識させるためにベンダID，プロダクトIDが必要です．今回は来栖川電工（有）のご厚意により，ベンダIDとして0x0B7E，プロダクトIDとして0x800Fを利用させていただけることになりました．この場を借りてお礼申し上げます．なお，このIDはあくまでも本稿向けの実験や動作確認用です．実際の製品などには絶対に使用しないでください．

エンドポイントバッファとの基本的なやりとりの手順は，次のようになります．

● EP0によるコントロール伝送

EP0によるコントロール伝送の手順は，おおむね次のようになります．

（1）SETUPパケットの到達検出

SETUPパケットが到達すると，UDC割り込みが発生します．USIR0レジスタのIR0がセットされるので，EP0からの割り込みであると判断されます．続いてUDCCS0レジスタを見ると，UDCCS0レジスタのSAビットとOPRビットが‘1’にセットされるので，SETUPパケットが到達していることがわかります．UDCCS0レジスタのSAビットがセットされていない場合には，単なるデータOUTなので，デバイスリクエストであると判断して処理すると，おかしなことになります．必ずSAビットをチェックしてください．

（2）SETUPパケットデータの取り出し

CPUがUDDR0レジスタ経由で到達したパケットデータを読み出します．他のOUTエンドポイントや，データOUTステージの伝送のときも同様ですが，このときUDCCS0レジスタのRNEビット（receiver not empty）がクリアされるまで読み続けることで，確実に全データを受け取ることができます．今回のサンプルでも，RNEビットを見ながらSETUPデータをバッファメモリに転送しています．

（3）SETUPパケット処理

SETUPパケットで受け取ったコマンドを処理します．

（4）データINが必要な場合

もし，SETUPデータへの応答としてデータINフェーズをともなう場合（GET_DESCRIPTORなど）には，UDDR0レジスタに1パケットサイズ分のデータを書き込み，ファームウェアはデータINステージを実行中であることがわかるようにします．今回のサンプルでは，USTATE_DATA_INがこれにあたります．

（5）UDC内の割り込みの始末

UDCCS0レジスタのSAビットとOPRビットをクリアします．データINフェーズをともなう場合にはさらにIPRビットもセットします．IPRビットがセットされると次のホストからのIN要求に対して，（4）のステップでUDDR0レジスタに書き込まれたデータがホストに送信されます．

（6）XScaleの割り込みの始末と復帰

USIR0レジスタのIR0ビットをクリアして割り込み処理から復帰します．

（7）EP0のデータIN割り込みへの応答

データINによる割り込みが発生したら，再びUDDR0レジ

スタにデータを書き込み，UDCCS0 レジスタの IPR ビットによってデータをセットし，(6)に移行します．

(8) ステータス OUT の処理

最終データまで送り終わると，ホストからサイズ0の OUT パケットが送られてきます(ステータス OUT フェーズ)．ターゲット側では UDCCS0 レジスタの OPR ビットが'1'で SA ビットが'0'であり，さらに動作ステートが USTATE_DATA_IN であることから，これがステータス OUT フェーズの OUT パケットであることがわかります．これを受信したら，ステートを IDLE 状態に戻し，UDCCS0 レジスタの OPR ビットと USIR0 レジスタの IR0 ビットをクリアしておきます．

● バルク IN/インタラプト IN

バルク IN とインタラプト IN 伝送の手順はそれほど変わりません．今回は，バルク IN はベンダリクエストでデータをセットさせるようにしているのに対して，インタラプト IN は常時スイッチデータを送信させるようにしているという使い方になっている点が異なる程度なので，ここではバルク IN のほうを例にとりあげることにしました．

(1) 割り込みマスク

何度かふれているとおり，XScale の UDC の割り込みはレベルトリガになっており，条件が成立している限り，何度でも割り込みが発生してしまいます．このため，IN 方向のエンドポイントバッファにセットすべきデータがない場合でもエンドポイントバッファが空で，エンドポイント割り込みがイネーブルになっていれば割り込みが発生し続けてしまいます．よって IN 方向で送るべきデータがない場合には，割り込みをマスクしておかなくてはなりません．

今回のファームウェアでは，エンドポイントごとに用意したコントロールテーブルの中に動作ステートフラグを用意して，IDLE 状態(送るものが何もない状態)のときには割り込みをマスクしてリターンさせるようにしています．

今回のサンプルでインタラプト IN，バルク IN とも起動後に割り込みを許可しているので，この時点で割り込みが入ってきますが，ステートが IDLE なのでエンドポイント割り込み処理の中で割り込みが禁止されることになります．

(2) 割り込み発生時の処理

EP1 のバッファに空きがあり，割り込みが発生した場合，USIR0 レジスタの IR1 がセットされるので，EP1 の処理ルーチンに分岐します．

(3) EP1(バルク IN エンドポイント)へのデータセット

IR1 がセットされていることがわかったら，バルク IN エンドポイント割り込みです．送るべきデータがあるなら，UDDR1 レジスタに必要なデータをセットします．もしセットするデータサイズがエンドポイントバッファサイズ(64 バイト)未満ならば，UDCCS1 レジスタの TSP ビットをセットして，UDC に対してデータセットが完了したことを通知します．

(4) EP1 割り込みの後始末

前回，バルク IN エンドポイントにセットしたデータが送信完了した結果の割り込みの場合，UDCCS1 レジスタの TPC ビットがセットされているので，こちらもクリアしておきます．さらに USIR0 レジスタの IR1 ビットもクリアして割り込み処理を完了させます．

(5) 全データ送信後

要求されたデータがすべて送信し終わった段階で割り込みが入ったときには，次の割り込みは不要なので割り込みをマスクしておきます．

● バルク OUT の処理

OUT 方向(ホストからターゲットの方向)の伝送は，基本的にパケットが到達したらそれを引きとり，次に備えるというだけになります．

(1) 割り込みの確認

EP2(バルク OUT)に OUT パケットが到達すると，USIR0 レジスタの IR2 ビットがセットされ，UDCCS2 レジスタの RPC ビットが'1'になるとともに割り込みが発生します．データが入っていれば UDCCS2 レジスタの RNE ビットが'1'になります．もし，UDCCS2 レジスタの RNE ビットが'0'でかつ RSP ビットが'1'であれば，Zero-LengthOUT パケットを受信したことを示します．

(2) データの引き取り

バルク IN で到達したデータの数を UBCR2 レジスタによって知り，その回数分読み出します．あるいは UDCCS2 レジスタの RNE ビットが'0'になるまで読み出すことで，全データを読み出すようにすることもできます．今回は後者の方法を使ってみました．

(3) 割り込みの後始末

到達したデータをすべて読み出し終わったら，UDCCS2 レジスタの RPC ビットをクリアして割り込みからリターンします．IN 方向と異なり，ホストから次の OUT パケットが到達するまで割り込みは発生しないので，IN 方向のエンドポイントのときのようなマスク処理などは不要です．

4 アプリケーションの作成

● ホスト側のサンプルアプリケーション

この記事は XScale 内蔵の USB 機能を使った USB ターゲットのプログラミング事例なので，Windows 側のドライバについては，参考文献1)で解説されている汎用 USB ドライバを使いました．また，サンプルアプリケーションについては Visual Basic5.0-CCE(評価版の VB)で作成しました．すでに VB の主流は.net 版に移行していますが，CCE 版はフリーでマイクロソフトのサイトから入手でき，機能的にも充分です．

● ドライバのインストール

USB デバイスを Windows 環境に接続するので，USB デバイ

スのインストールが必要です．参考文献1)の説明にしたがい，INFファイルを用意します．すでに説明したように，ベンダIDは0x0B7E，プロダクトIDは0x800Fです．

ファイルが完成したら，評価ボードとホストPCをシリアルケーブルで接続します．ただし，USBケーブルはまだ接続しないでください．

評価ボードの電源を入れ，WATCHPOINTデバッガを起動します．プロジェクトを作成して，作成したUSBターゲットのサンプルプログラムをダウンロードします．ダウンロードが完了したら，プログラムの実行を開始します．

プログラムの実行が開始されると，評価ボード上のLEDが消えるので，ここでUSBケーブルを接続します．するとホスト側にデバイスが認識されるので，先ほど作成したINFファイルにより汎用USBドライバをインストールします．

● サンプルアプリケーションの実行

この状態でVBアプリケーションを起動すると，**図10(a)**に示すような画面が表示されます．上に並んだ8ビット分のチェックボックスで点灯させたいLEDのセグメントを指定してSET_LEDボタンを押すと，バルクOUT経由でデータが送られて評価ボード上のLEDの点灯状態が変化します．

二段目はバルクINによるディップスイッチの状態で，GET_DIPSWボタンを押すと現在のディップスイッチの状態が読み出されます．

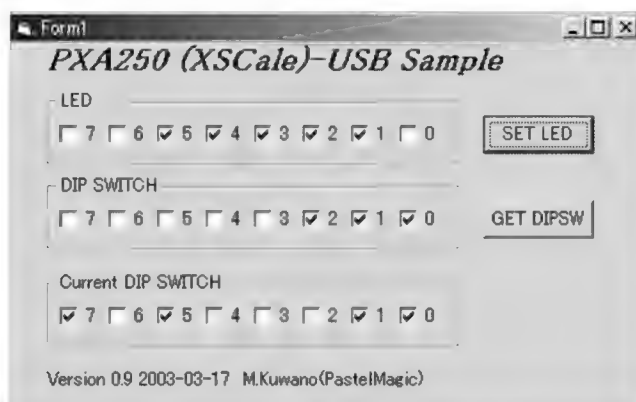
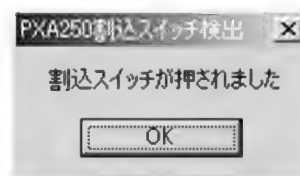
いちばん下の8ビット分はインタラプトINで読み出したディップスイッチのデータで，タイマによって自動的に一定周期で更新されます．

図10(a)の画面で，バルクINのデータとインタラプトINのデータが違っているのは，バルクINで読み出した後でディップスイッチを操作したためです．インタラプトINでは一定周期でディップスイッチの状態を取得しますが，バルクINはGET_DIPSWボタンを押したときに状態を取得するからです．

また，評価ボードに取り付けた割り込みスイッチを押すと，インタラプトINデータで情報が取り込まれて，図10(b)のように割り込み発生のダイアログボックスが出るようにしています．

これで基本的な通信はできるようになったので，あとはファームウェアやVBアプリケーションをいろいろいじってみるとよいでしょう．

最後に，ここで作成したUSBターゲットのサンプルプログラムやVB用サンプルプログラムのソースは，本号付属CD-ROMに収録しています．

〔図10〕ホスト側サンプルアプリケーション

(a) 動作画面

(b) 割り込み発生時

まとめ

今回初めてXScaleのUSBファンクションを動かしてみました．機能的にあまり欲張ったところがない分，シンプルにできていますが，レベルトリガを基本にすえた割り込み関係やレジスタのステータスやコントロールビットの取り扱いについては少々クセがあり，注意が必要でした．エンドポイントの数は比較的豊富なので，さまざまな用途に展開できる可能性をもったインターフェース仕様であるといえるでしょう．

なお，デバッガホストとの接続インターフェースにUSBを使う場合は，当然ながら今回のプログラムは同時には使用できません．USB接続ではユーザープログラムのダウンロードなどが高速に行えるので，デバッグ時にはこちらを使いたいのですが，ユーザーアプリケーションでUSBを使いたいとなると，USB接続はできません．CPUにはUSB機能が一つしかないので，これはいたしかたないところでしょう．

参考文献

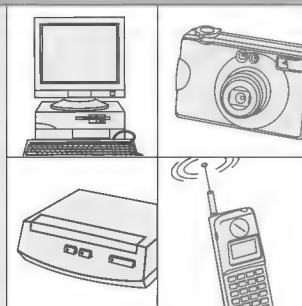
1)『USBハード＆ソフト開発のすべて』，TECH I Vol.8，CQ出版(株)

くわの・まさひこ　パステルマジック

家電機器をネットワーク化するアーキテクチャ
Universal Plug and Playの全貌

第3回 Windows XPでのUPnPプログラミング

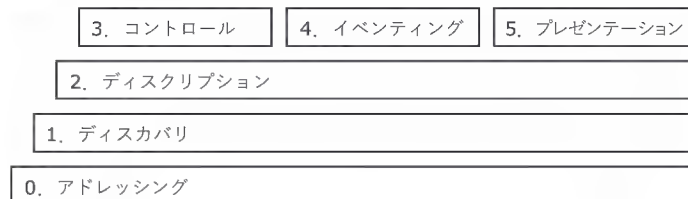
長尾 康

情報家電をはじめとした機器をネットワークへ手軽に接続するための手段として，Universal Plug and Playが注目されている．前回までの連載では，Universal Plug and Playについて，その規格概要について解説した．連載第3回目の今回は，Windows XPにおけるUniversal Plug and Playのプログラミングについて，解説を行う．（編集部）

これまで2回にわたりUniversal Plug and Playデバイスアーキテクチャ規格の解説があり，規格の内容は理解できたと思います．そこで今回は，実践的な内容として，Windows XPのUPnPの機能を紹介し，マイクロソフトから提供されているMicrosoft Platform SDKのサンプルを用いてプログラミングする方法を説明します．また，IGD（Internet Gateway Device：日本ではUPnPルータと呼ばれている）をコントロールする方法について，サンプルを交えながら紹介します．

Microsoft Platform SDK

Platform SDKは，MSDNサブスクリプションを購入することで入手できます．MSDNのサブスクリプションはレベルによって違いますが，開発に必要なマイクロソフトのソフトウェア，ドキュメントおよびOSを入手できます．また，購入後，Webページからのダウンロードも可能になります．詳細は，以下のURLを参照してください．

http://www.microsoft.com/japan/msdn/subscriptions/default.asp

〔図1〕UPnPのステップ

Windows XPにおけるUPnPの概要

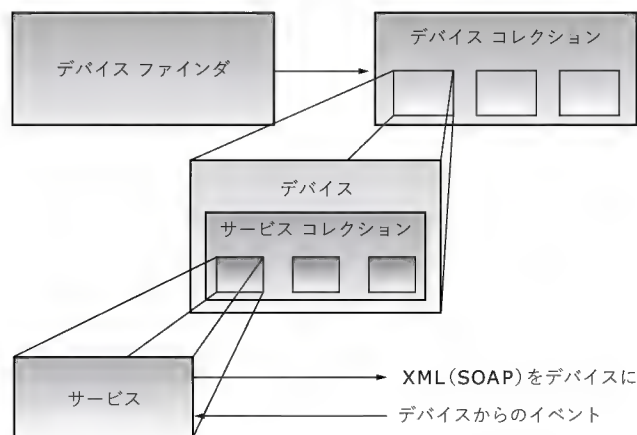
Windows XPには，コントロールポイントのアプリケーションを動作させるための機能とUPnPデバイスとして動作させる機能があります．Platform SDKでは，コントロールポイントAPIおよびデバイスホストAPIと定義されています．これらの機能は，COM（Component Object Model）で実装されています．そのため，COMの知識が必要です．もしCOMに関する知識がない場合は，以下の解説がわかりやすいでしょう．あとは，SDKのドキュメントとサンプルを参考に学んでください．

“Dr. GUI，コンポーネント，COM，およびATLを使う”
http://www.microsoft.com/japan/msdn/library/default.asp?url=/japan/msdn/library/ja/jpdnguion/htm/msdn_drguion020298.asp

なお本記事では，オブジェクトという言葉は使用しませんでした．意味の幅が非常に広く，あいまいになりがちだからです．COM関連のリソースはコンポーネント，インターフェース，メソッドという言葉を使用しました．実際にはオブジェクトと表現するほうが適切な箇所もあるかと思いますが，あえて統一しました（ちなみにPlatform SDKのヘルプでは，オブジェクトという言葉を使っている）．

Windows XPは，前回までの説明にあった**図1**にある機能を備えています．その上部にコントロールポイントCOMレイヤとデバイスホストCOMレイヤ（**図2**）があります．それぞれコントロールポイントAPIおよびデバイスホストAPIをもっており，デバイスをコントロールするアプリケーションはコントロールポイントAPIを使用します．デバイスとして動作させるアプリケーションは，デバイスホストAPIを使用します．

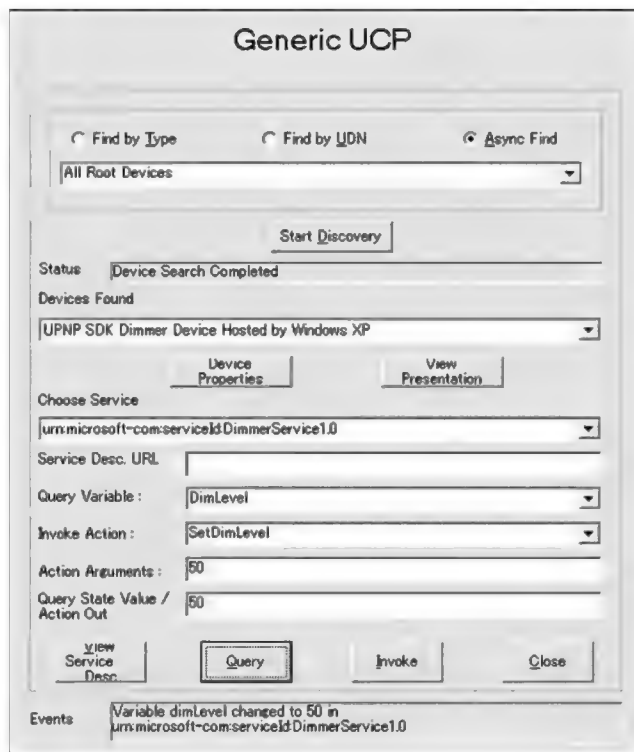
〔図2〕コントロールポイントCOMレイヤとデバイスホストCOMレイヤ

〔図3〕コンポーネントの構成

コントロールポイントAPI

コントロールポイントを構築するためのAPIは，どのような機能をもっているのでしょうか？ 当然ですが，UPnPデバイスアーキテクチャのステップと同じようにデバイスの検出，デバイスディスクリプションの取得，デバイスのコントロールを行います．以下に，コントロールポイントを構築するためのAPI(COMでは，インターフェースと呼ぶ)の代表的なものを紹介します．ほかのインターフェースは，非同期アクセスをサポートするため，もしくは補助的なものです．SDKのヘルプファイルを参照してください．

- `IUPnPDeviceFinder`
  このインターフェースによって，アプリケーションがデバイスを検出できます．
- `IUPnPDevices`
  アプリケーションがデバイスを列挙します．
- `IUPnPDevice`
  アプリケーションがデバイスについての情報を収集できます．
- `IUPnPServices`
  アプリケーションがデバイス内にあるサービスを列挙します．
- `IUPnPService`
  アプリケーションがサービスのインスタンス上で変数を検索，アクションを実行することができます

次に，Windows XPのUPnP機能がUPnPデバイスをどのようにオブジェクト定義しているかを説明します．これはプログラミングやそれにともなうCOMインターフェースがどのように継承されるかに関わるため，重要です．

では，UPnPデバイスアーキテクチャの規格を思い出してください．デバイスの構造は簡単に説明すると，ルートデバイスというコンテナがあり，その中に複数のデバイス，そしてデバイスの中に複数のサービスがあったと思います．**図3**を参照してください．コントロールポイントCOMレイヤは，デバイスファインダというコンポーネントでデバイスを管理します．このコンポーネントはデバイスを検出しますが，検出方法によってはデバイスが複数検出されます．各デバイスにはサービスがあり，このサービスのコンポーネントがコントロールを実行します．

なお，**図3**ではシンプルな例になっていますが，実際にはデバイスコレクション内はツリー構造になっています．これらコンポーネントモデルにしたがってデバイスファインダのインターフェースは各コンポーネントへ継承されていきます．

〔図4〕サンプルプログラムの画面

サンプルの説明

Platform SDKをインストールすると，UPnPのサンプルもインストールされます．デフォルトでは“ドライブ名:`¥Program Files¥Microsoft SDK¥Samples¥netds¥upnp¥genericucp`”に，コントロールポイント用のGeneric Control Pointというサンプルがインストールされます．C++とVBの二つサンプルがあります．言語が違うだけで同じプログラムですが，C++のサンプルにはView Service DescとView Presentationの機能が実装されていません．しかし今回は，実用性などを考えてC++のサンプルを解説します．**図4**が，後半で説明する仮想デバイスである調光器を接続した例です．`SetDimLevel`によって変数の値を50にセットした例です．`DimLevel`を`Query`すると`Query State Value`の値が50になっています．

次に，コントロールポイントアプリケーションプログラムの処理法を，**図1**のステップにしたがって説明します．

● アドレシッング

この部分は，Windows XPのネットワークスタックが自動的に処理するので，UPnPのAPIは関係ありません．

〔**リスト1**〕**GenericUCPDlg.cpp，594行目〜**

```
// Instantiate the device finder object
hr = CoCreateInstance( CLSID_UPnPDeviceFinder,
                       NULL,
                       CLSCTX_SERVER,
                       IID_IUPnPDeviceFinder,
                       (LPVOID *)&m_pDeviceFinder);
```

〔**リスト2**〕**GenericUCPDlg.cpp，824行目〜**

```
void CGenericUCPDlg::ProcessFindByUDN(BSTR bstrSearchType)
{
    // FindByUDN selected
    HRESULT hr= S_OK;

    m_StatusText.SetWindowText(_T("FindByUDN selected"));
    ASSERT(m_pDeviceFinder);
    IUPnPDevice* pDevice = NULL;
    hr = m_pDeviceFinder->FindByUDN(bstrSearchType, &pDevice);
    if(hr==S_OK){
        // Found a device with given udn
        // Add the found device to the device list
        // Get the friendly name of the device
        BSTR bstrFriendlyName = NULL;
        hr = pDevice->get_FriendlyName(&bstrFriendlyName);
        if(SUCCEEDED(hr)){
            TCHAR tszFriendlyName[DATA_BUFSIZE];
            _sntprintf(tszFriendlyName, DATA_BUFSIZE-1, _T("%S"),
                                                   bstrFriendlyName);
            m_DeviceCombo.AddString(tszFriendlyName);
            m_DeviceCombo.SetItemDataPtr(0, pDevice);
```

● ディスカバリ

どのようにデバイスを検出するか説明します．具体的には，**図3**に示すようにどのようにコンポーネントをたどっていくかの手順を説明します．

1）デバイスファインダコンポーネントのインスタンスを生成します．これによって，コントロールポイントCOMレイヤのインターフェース（API）をアプリケーションから実行できます．サンプルの一部（**リスト1**）を参照してください．

2）次に，デバイスの検出を行います．検出方法は，デバイスそのものを直接検出する方法，もしくは関連するデバイスすべて（デバイスコレクション）を検出する方法があります．また，同期と非同期によるAPIの実行方法がありますが，今回は同期による方法だけを解説します．非同期の特徴としては，検出を開始した後，ネットワークに追加されるようなデバイスがあった場合に見逃さずハンドルできる点です．

●特定のデバイス（UDN）によって指定する方法（**リスト2**）

UDNは，デバイスを示す唯一の名前になります．このため，直接デバイスコンポーネントのインターフェースが渡ってきます．&pDeviceを見てほしいのですが，これはIUPnPDeviceインターフェースのポインタになります．この例では，IUPnPDeviceのget_FriendlyNameしか実行していませんが，詳細はディスクリプションの中で説明します．

●デバイスのタイプ（URI）によって指定する方法（**リスト3**）

この方法は，同じタイプのデバイスが複数ある場合，すべてのデバイスを検出します．そのときにはデバイスコレクションになります．&pDevicesに注目してください．これは，

〔**リスト3**〕**GenericUCPDlg.cpp，681行〜**

```
void CGenericUCPDlg::ProcessFindByType(BSTR bstrSearchType)
{
   // We have to process FindByType search
   HRESULT hr = S_OK;
   IUPnPDevices* pDevices = NULL;
   IUnknown* punkEnum = NULL;
   IEnumUnknown *pEU = NULL;

   ASSERT(m_pDeviceFinder);
   m_StatusText.SetWindowText(_T("FindByType selected"));
   hr = m_pDeviceFinder->FindByType(bstrSearchType, 0,
                                                   &pDevices);
   if (SUCCEEDED(hr))
   {
      // We have to enumerate the devices
      long lCount;
      hr = pDevices->get_Count(&lCount);
      if (SUCCEEDED(hr))
      {
         if (lCount != 0)
         {
            hr = pDevices->get__NewEnum(&punkEnum);
            if (SUCCEEDED(hr))
            {
               hr = punkEnum->QueryInterface(IID_IEnumUnknown,
                                             (VOID **) &pEU);
               if (SUCCEEDED(hr))
               {
                  for (long lIndex = 0; lIndex<lCount; lIndex++)
                  {
                     IUnknown* punkDevice = NULL;
                     IUPnPDevice* pDevice=NULL;
                     hr = pEU->Next(1, &punkDevice, NULL);
                     if (SUCCEEDED(hr))
                     {
                        // Get a IUPnPDevice pointer to the
                                          device just got
                        hr = punkDevice->QueryInterface(
                                      IID_IUPnPDevice,
                                      (VOID **)&pDevice
                                      );
                        if (SUCCEEDED(hr))
                        {
                           // Add the found device to the device
                                                          list
                           // Get the friendly name of the
                                                        device
                           BSTR bstrFriendlyName = NULL;
                           hr = pDevice->get_FriendlyName(
                                          &bstrFriendlyName);
                           if (SUCCEEDED(hr))
                           {
                              TCHAR tszFriendlyName[
                                            DATA_BUFSIZE];
                              _sntprintf(
                                 tszFriendlyName,
                                 DATA_BUFSIZE - 1,
                                 _T("%S"),
                                 bstrFriendlyName
                                 );
                              m_DeviceCombo.AddString(
                                          tszFriendlyName);
                              m_DeviceCombo.SetItemDataPtr(
                                          (int)lIndex,
                                          pDevice
                                          );
                              SysFreeString(bstrFriendlyName);
```

IUPnPDevicesインターフェースのポインタになります．つまり，デバイスコレクションになります．デバイスコンポーネントそのものでないので，ここからデバイスコンポーネント(IUPnPDevice)インターフェースを列挙する必要があります．

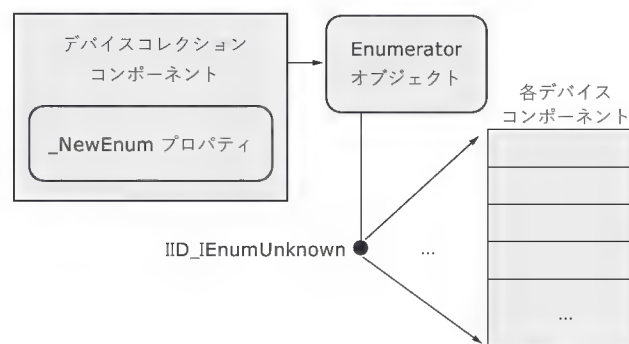
ここからは，このデバイスコレクションからデバイスコンポーネントを探る方法を説明します．まず，IUPnPDevicesコンポーネントは，ActiveXのコレクションオブジェクトと同じ設計になっています(**図5**)．ですから，_NewEnumプロパティを利用して，Enumeratorオブジェクト取得します．

```
hr = pDevices->get__NewEnum(&punkEnum);
```

次に，IID_IEnumUnknownインターフェースのオブジェクトを取得します．IID_IEnumUnknownのNextメソッドを実行し，デバイスコレクション内の最初のデバイスコンポーネントを取得します．これでやっとデバイスコンポーネントにたどり着けたわけです．ここで&pDeviceを見ると，IUPnPDeviceインターフェースのポインタになっています．

● ディスクリプション

1) アプリケーションプログラムがデバイスディスクリプション情報を取得するには，IID_IUPnPDeviceインターフェースがもっているメソッドを実行します．**表1**にメソッド一覧を挙げます．デバイスコンポーネントを取得した時点でWindows XPのUPnPスタックは，すでにデバイスディスクリプションを取得しているようです．これらメソッドを使うことにより，その内容を確認できます．また，これらによってデバイスの属性やツリー構造を確認できますが，その中でももっとも重要なのが，Servicesメソッドです．このメソッドによってサービスコレクションのコンポーネントを取得します．

2) デバイス内のサービスを検出する場合は，サービスタイプを指定してダイレクトに調べることができません．**図3**にあったように，デバイスコンポーネントからサービスコレクションを取得します．&pServicesに注目してください．これは，IUPnPServicesインターフェースのポインタになります．サービスコンポーネントそのものでないので，ここからサービスコンポーネント(IUPnPService)インターフェースを列挙する必要があります．このサービスコレクションからサービスコンポーネントを探るソースコードを**リスト4**に示します．IUPnPServicesコンポーネントは，デバイスコレクションと同じコンポーネント構成になるのでそちらをご参照ください(**図5**)．ここで&pServiceを見るとIUPnPServiceインターフェースのポインタになっています．また，アプリケーションプログラマが目的のSerivceIDをわかっていれば，サービスオブジェクトを列挙することなく，次のようにDVDVideoというサービスコンポーネントを取得できます．

```
BSTR bstrServiceName = SysAllocString(
```

〔図5〕IUPnPDevicesコンポーネント

〔表1〕IID_IUPnPDeviceインターフェースのメソッド

メソッド	機　能
IsRootDevice	指定したデバイスオブジェクトがルートデバイスかどうか調べる
RootDevice	指定したデバイスオブジェクトのルートデバイスのデバイスオブジェクトを取得する
ParentDevice	指定したデバイスオブジェクトの親デバイスのデバイスオブジェクトを取得する
HasChildren	指定したデバイスオブジェクトが入れ子のデバイスを持っているか調べる
Children	入れ子のデバイスオブジェクトを取得する
UniqueDeviceName	Unique device name(UDN)を取得する
FriendlyName	デバイスの表示名を取得する
Type	デバイスタイプを示すUniform resource identifier(URI)を取得する
PresentationURL	デバイスが用意しているWebページのURL(Presentation URL)を取得する
ManufacturerName	製造社名をUnicodeで取得する
ManufacturerURL	製造社のURLを取得する
ModelName	モデル名をUnicodeで取得する
ModelNumber	モデルナンバをUnicodeで取得する
Description	デバイスの機能の概要をUnicodeで取得する
ModelURL	デバイスのモデルに関する情報が紹介されているURLを取得する
UPC	デバイスの製造コードをUnicodeで取得する
SerialNumber	デバイスのシリアルナンバをUnicodeで取得する
IconURL	デバイスを検出し表示するときに使用するIconデータのURLを取得する
Services	デバイスが持っているサービスのリストを示すIUPnPServicesインターフェースオブジェクトを取得する

```
        L"urn:upnp-org:servicId:DVDVideo");
    IUPnPService * pAppService = NULL;
    hr = pServices->get_Item
                    (bstrServiceName,&pAppService);
```

ここまでが，UPnPデバイスアーキテクチャ規格のアドレッシング，ディスカバリ，ディスクリプションの処理です．アプリケーションプログラムでは，サービスコンポーネントを検出するまでの作業になります．**図6**に示すように，デバイスをコントロールするのは，その下位の部分になります．コントロール，イベント，プレゼンテーションは，ディスクリプションまでの処理が終わっていれば，相互に関係なく実行できます．ここからサービスコンポーネントのインターフェースを使ってデバイスをコントロールします．

● コントロール

アクションの実行，および変数の値を取得するなどのサービスインターフェースのメソッドを実行します．通常，プログラマは，プログラムによってどのようなことをするのか決めているので，実行するアクションをはじめからプログラムに組み込めます．そのため，サービスがどのようなアクションや変数をもっているかについて調べる必要がありません．よって，サービスインターフェースにアクション名や変数名を調べるメソッドはありません．ユーティリティツールなどを作る場合は，IUPnPDescriptionDocumentやIUPnPDeviceDocumentAccessを使用し，サービスディスクリプションのURLを取得して，直接サービスディスクリプションの内容を調べる必要があります．

1）アクションの実行

アクション名と変数へ希望する値を指定し，InvokeActionを実行します（**リスト5**）．

2）変数値のクエリ

変数の値を調べるときは，サービスインターフェースのQueryStateVariableメソッドを使用します．このときに変数名を指定します（**リスト6**）．

● イベント

イベントは外的要因でサービスの変数の値が変わったときに，コントロールポイントへサービスが通知するもので，サブスク

〔リスト4〕GenericUCPDlg.cpp，1266行目～

```
hr = pDevice->get_Services(&pServices);
if(hr==S_OK){
    long lCount;
    hr = pServices->get_Count(&lCount);
    if(SUCCEEDED(hr)){
        if(lCount!=0){
            // We have to get a IEnumUnknown pointer
            hr = pServices->get__NewEnum(&punkEnum);
            if(SUCCEEDED(hr)){
                hr = punkEnum->QueryInterface(IID_IEnumUnknown, (VOID **) &pEU);
                if(SUCCEEDED(hr)){
                    for(lIndex = 0; lIndex<lCount; lIndex++){
                        IUnknown *punkService = NULL;
                        IUPnPService *pService=NULL;
                        hr = pEU->Next(1, &punkService, NULL);
                        if(SUCCEEDED(hr)){
                            // Get a IUPnPService pointer to the service just got
                            hr = punkService->QueryInterface(IID_IUPnPService, (VOID **)&pService);
                            if(SUCCEEDED(hr)){
                                BSTR bstrServiceId = NULL;
                                hr = pService->get_Id(&bstrServiceId);
                                if(SUCCEEDED(hr)){
                                    TCHAR tszServiceId[DATA_BUFSIZE];
                                    _sntprintf(tszServiceId, DATA_BUFSIZE-1, _T("%S"), bstrServiceId);
                                    m_ServiceCombo.AddString(tszServiceId);
                                    m_ServiceCombo.SetItemDataPtr((int)lIndex, pService);
                                    SysFreeString(bstrServiceId);

}
```

〔図6〕UPnPアーキテクチャの構成

〔リスト5〕GenericUCPDlg.cpp，1627行目～

```
hr = pService->InvokeAction( bstrActionName,
                             vaActionArgs,
                             &vaOutArgs,
                             &vaRet);
if(SUCCEEDED(hr))
```

〔リスト6〕GenericUCPDlg.cpp，1472行目～

```
hr = pService->QueryStateVariable(bstrVariableName, &varValue);
```

〔リスト7〕GenericUCPDlg.cpp，1965行目

```
pCallback->CreateInstance(&pCallback);
```

〔リスト8〕GenericUCPDlg.cpp，1379行目

```
hr = pService->AddCallback(m_pServiceCallback);
```

ライブが必要です．Windows XPでは，サービスインターフェースのAddCallBackメソッドを使用してIUPnPServiceCallbackインターフェースをコントロールポイントAPIへ登録します．変数の値が変更されたときにIUPnPServiceCallbackのStateVariableChangedメソッドがコントロールポイントAPIから呼び出されます．また，サービスのインスタンスや登録していたイベントが無効になったときは，IUPnPServiceCallbackのServiceInstanceDiedが実行されます．

以下のように，プログラム中でIUPnPServiceCallbackインターフェースを追加して登録します．サンプル中では，**リスト7**の箇所でインターフェースを生成し，**リスト8**の箇所で登録しています．ここで変更された変数と値を受けて，それらを**リスト9**で表示しています．**リスト10**は，サービスのインスタンスがなくなったときに呼び出されます．

● プレゼンテーション

プレゼンテーションは簡単です．デバイスインターフェース(IUPnPDevice)のPresentationURLメソッドを実行して入手したURLを元にWebブラウザ(IEなど)を実行します．今回のサンプルにはインプリメントされていません．

デバイスホストAPI

Windows XPをデバイスとして動作させるためのAPIです．じつはWinHEC 2003において，Content Directory Servicesという新しい機能の発表がありました．これは，Windows XPのマイミュージック，マイビデオやマイピクチャというディレクトリにあるコンテンツをネットワークへ開放し，ステレオなどのオーディオ機器やTV(STBも含む)でネットワークを介して再生するものです．これはUPnP AVスペック(UPnP Media Server)に準拠しています．これらはWindows XP上でUPnPデバイスとして動作します．また，Windows XPでインターネット接続サービス(ICS)を設定するとUPnP IGD(インターネットゲートウェイデバイス)の機能をサポートします．これもUPnPデバイスとして動作しています．

では，Windows XPのこのデバイスホストAPIレイヤの機能概要を説明します．デバイスをWindows XP上に実装するためにデバイスホストAPIレイヤは，以下の機能を備えています．

- Windows XPへ登録されたデバイスの存在をアナウンス
- アドバタイズ(告知)を自動更新

〔リスト9〕GenericUCPDlg.cpp，2008行目～

```
HRESULT CServiceCallback::StateVariableChanged(
    IUPnPService *pus,
    LPCWSTR pcwszStateVarName,
    VARIANT varValue)
{
    HRESULT hr = S_OK;
    TCHAR tszMessage[DATA_BUFSIZE];
    TRACE(_T("State Variable Changed¥n"));

    BSTR bstrServiceId = NULL;
    hr = pus->get_Id(&bstrServiceId);
    if(SUCCEEDED(hr)){
        TCHAR tszServiceId[DATA_BUFSIZE];
        _sntprintf(tszServiceId, DATA_BUFSIZE-1, _T("%S"),
                                                bstrServiceId);
        SysFreeString(bstrServiceId);
        hr=VariantChangeType(&varValue, &varValue,
                                VARIANT_ALPHABOOL, VT_BSTR);
        if(SUCCEEDED(hr)){
            _sntprintf(tszMessage, DATA_BUFSIZE-1,
                    _T("State variable %S changed to %S in %S"),
                    pcwszStateVarName, varValue.bstrVal,
                    bstrServiceId);
            Sleep(350);
            m_pGenericUCPDlg->m_EventText.SetWindowText(
                                                tszMessage);
        }
        else{
            PrintErrorText(hr);
        }
    }
    else{
        PrintErrorText(hr);
        m_pGenericUCPDlg->m_StatusText.SetWindowText(
                            _T("Error: ServiceId failed"));
    }
    return hr;
}
```

〔リスト10〕GenericUCPDlg.cpp，2055行目～

```
HRESULT CServiceCallback::ServiceInstanceDied(
    IUPnPService *pus)
{
    HRESULT hr=S_OK;
    TCHAR tszMessage[DATA_BUFSIZE];
    TRACE(_T("Service instance died¥n"));

    BSTR bstrServiceId = NULL;
    hr = pus->get_Id(&bstrServiceId);
    if(SUCCEEDED(hr)){
        _sntprintf(tszMessage, DATA_BUFSIZE-1,
                _T("Service %S died"), bstrServiceId);
        m_pGenericUCPDlg->m_EventText.SetWindowText(tszMessage);
        SysFreeString(bstrServiceId);
    }
    else{
        PrintErrorText(hr);
        m_pGenericUCPDlg->m_StatusText.SetWindowText(
                            _T("Error: ServiceId failed"));
    }
    return hr;
}
```

- サーチ(検索)要求にも応答
- デバイス/サービスディスクリプションの要求に応答
- コントロールポイントの要求でサービスのアクションを起動
- イベントを処理
- サブスクライブリクエストの受け入れ
- サービスごとにサブスクライバの一覧を管理
- すべてのサブスクライバにイベントを送信(変数が変化したとき)

では，実際のデバイスホストAPIとデベロッパが開発すべき

モジュールの関連を**図7**に示します．コントロールポイントはイベントなどの処理でコールバックがあるものを除き，アプリケーションをユーザーが操作することでコントロールポイントAPIを使用し，UPnPの機能を実行しています．いたってシンプルな構造です．しかしデバイスの実装は，デバイスの設定などを除き，ほとんどユーザーが操作することはありません．すべてコントロールポイントからの要求をUPnPスタック→ホストデバイスAPIを経由して実行します．

デバイスの実装

デベロッパは**図7**の左側の“デバイスの実装”と書かれたモジュールを開発します．このモジュールは，デバイスホストAPIのもっているインターフェースを使用し，Windows XPのUPnPスタックの機能を使用します．またデバイスホストは，告知やコントロールポイントからの要求時に，このモジュールのインターフェースを呼び出しデバイスの機能を実行します．

では，デベロッパは，デバイスを実装する場合，どのようなものを開発，用意する必要があるのでしょうか？　以下にそれを示します．

- XMLによるデバイスディスクリプションテンプレートを用意
- 各サービスにSCPD XMLファイルを用意
- `IUPnPDeviceControl`インターフェースを公開するCOMコンポーネントを実装
- 各サービスコンポーネントに`IDispatch`インターフェースを実装
- デバイスをUPnPインターフェースへ公開するために`IUPnPRegistrar`インターフェースを実行

〔図7〕各モジュールの関連

〔図8〕二つのPCの役割

サンプルの説明

最初に，今回解説に使用するサンプルの説明をします．このSDKのサンプルは，調光器を想定した仮想デバイスです．**図7**の構成に照らし合わせると“デバイスの実態”は`UPNPSampleDimmerDevice.dll`になり，デバイスディスクリプションテンプレートは，`DimmerDevice-Desc.xml`，サービスディスクリプション(SCPD)は，`DimmingService_SCPD.xml`になります．また，この調光器をデバイスホストに登録，起動，公開するプログラムは，`RegDevice.exe`です．ビルドの方法は，SDKをご覧ください．

使用方法を説明します．まず，ネットワークで2台のPC(もちろんWindows XP)をつなぎ，一つはコントロールポイントとして，もう一つを調光器として使用します(**図8**)．

SDKのインストール先を変更しなければ，サンプルの場所は以下のとおりです．

- 調光器を想定した仮想デバイス
  ドライブ名:`¥Program Files¥Microsoft SDK¥Samples¥netds¥upnp¥DCO_DimmerService`
- 調光器を登録，起動するソフトウェア
  ドライブ名:`¥Program Files¥Microsoft SDK¥Samples¥netds¥upnp¥RegisterDevice`

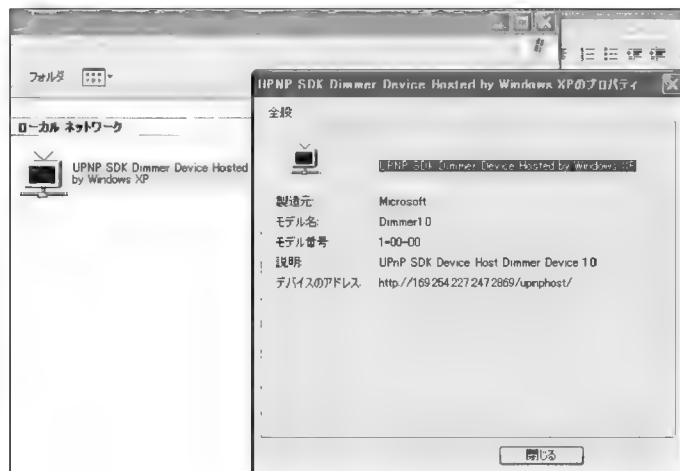
まず，調光器のPCのコマンドプロンプトから`regsvr32`ツールによって，`UPNPSampleDimmerDevice.dll`(COM)コンポーネントをシステムへ登録します(**図9**)．

次に`RegDevice.exe`を実行し，調光器を公開します(**図10**)．このとき`DimmerDevice-Desc.xml`と`Dimming`

〔図9〕コンポーネントをシステムへ登録する

```
dir
 ドライブ C のボリューム ラベルがありません。
 ボリューム シリアル番号は 8C5B-CEE7 です

 C:¥Program Files¥Microsoft SDK¥Samples¥netds¥upnp¥DCO_DimmerService¥XP32_RETAIL のディレクトリ

2003/04/13  16:49    <DIR>          .
2003/04/13  16:49    <DIR>          ..
2003/04/13  16:49           108,188 devicedll.obj
2003/04/13  16:49            40,669 dimmerdevicedco.obj
2003/04/13  16:49            24,301 dimmerservice.obj
2003/04/13  16:49            33,280 UPNPSampleDimmerDevice.dll
2003/04/13  16:49             1,014 UPNPSampleDimmerDevice.exp
2003/04/13  16:49             1,710 UPNPSampleDimmerDevice.lib
2003/04/13  16:49             5,236 UPNPSampleDimmerDevice.res
               7 個のファイル             214,398 バイト
               2 個のディレクトリ  25,491,025,920 バイトの空き領域

C:¥Program Files¥Microsoft SDK¥Samples¥netds¥upnp¥DCO_DimmerService¥XP32_RETAIL>
regsvr32 UPNPSampleDimmerDevice.dll

C:¥Program Files¥Microsoft SDK¥Samples¥netds¥upnp¥DCO_DimmerService¥XP32_RETAIL>
```

〔図10〕調光器を公開する

〔図11〕マイネットワークにアイコンが表示される

〔リスト11〕DimmerDevice-Desc.xml

```
<serialNumber>0000001</serialNumber>
  <UDN>uuid:RootDevice</UDN>
  <UPC>00000-00001</UPC>
  <serviceList>
    <service>
      <serviceType>urn:microsoft-com:service:DimmerService:1
                                                    </serviceType>
      <serviceId>urn:microsoft-com:serviceId:DimmerService1.0
                                                      </serviceId>
      <controlURL />
      <eventSubURL />
      <SCPDURL>DimmingService_SCPD.xml</SCPDURL>
    </service>
  </serviceList>
  <presentationURL />
  </device>
</root>
```

Service_SCPD.xmlをRegDevice.exeと同じディレクトリに用意しておきます．

RegDevice.exeを実行すると，コントロールポイントのPCに**図11**のようにマイネットワークにアイコンが出てきます．これを確認できればデバイスが動作していると判断できます．

1）デバイスディスクリプションテンプレートの作成

デバイスディスクリプションテンプレートを作成します．デバイスディスクリプションにURLなどを記述するアイテムがありますが，それらの一部はデバイスホストが動作している状況（IPアドレス含め）に合わせてセットされ，ディスカバリやサーチのときにデバイスホストがセットした内容を含めたデバイスディスクリプションをコントロールポイントへ送信します．まず，**リスト11**にあるデバイスディスクリプションテンプレートの例の一部をご覧ください．

- UDNはデバイスホストに登録されるとデバイスホストがUDNを生成します．
- ControlURL，eventSubURL,に関しては，何も記述しないでください．デバイスホストへ登録されるとデバイスホストがURLを用意します．
- SCPDURL，presentationURL，サンプルにはありませんがiconのURLも記述しません．ただし，それぞれのファイル名をセットしてください．xml，html，Iconファイルになります．

2）SCPD（サービスディスクリプション）の作成

SCPDのUPnP規格にしたがってサービスの機能を示すように記述してください．

3）デバイスコンポーネントの実装

デバイスコンポーネントを実装する場合，必ず，デバイスコントロールインターフェース（IUPnPDeviceControlインターフェース）を用意します．デバイスコンポーネントがデバイスホストへ登録されたときに，デバイスホストへこのインターフェースが渡されます．デバイスホストがUPnPのネットワーク環境のインターフェースになります．言い換えるとコントロールポイントはデバイスホストを窓口にデバイスコンポーネントにアクセスすることになります．そのときにデバイスホストがデバイスコンポーネントを認識するためのインターフェースがIUPnPDeviceControlなのです．また，実際にコントロールポイントからの個々のリクエストを処理するのはサービスコンポーネントです．当然，デバイスコンポーネントは，サービスコンポーネントを実装します．なお，サンプルはATLを使用しています．

4）サービスコンポーネントの実装

サービスコンポーネントはディスパッチインターフェース（IDispatch）とイベントソースインターフェース（IUPnPEventSource：これに関しては，ソースコードの中で説明する）を検討しなければなりません．デバイスホストは，コントロールポイントからのコントロールの実行やイベントの購読要求を受け取ります．デバイスホストは，それら要求から適切なサービスコンポーネントのIDispatchインターフェース/メソッドを呼び出します．反対にデバイスコンポーネントは，デバイスホストの呼び出しに合わせてIDispatchインターフェースのメソッドを用意する必要があります．

それでは，どのようにメソッドを用意するか説明します．まず，デバイスホストは，デバイス/サービスディスクリプションを呼び込み，デバイスディスクリプションを構築し，サービスディスクリプション(アクション，イベント)から，IDispatchインターフェースのメソッドを呼び出すためのテーブルを用意します．つまり，そのテーブルにあわせてサービスコンポーネントのメソッドを実装すればよいわけです．どのように実装するかというと，COMの場合インターフェースやメソッドの構成をIDLファイルで定義します．ホストデバイスのようにサービスディスクリプションからIDLを作るのがよいでしょう．簡単にUTL(UPnP Template Language)からIDLを生成するツールがPlatform SDKに用意されています．

では，**図12**のようにコマンドプロンプトでutl2idl.exeを実行します．DimmingService_SCPD.idlが生成されたと思います．ただし，"ドライブ名:¥**********s¥upnp¥DCO_DimmerService"にあるDimmerDevice.idlと比べるといろいろな項目が抜けていると思いますが，COM，ATLのルールにしたがって追加する必要があります．最初のほうで紹介したMSDNの"Dr. GUI，コンポーネント，COM，およびATLを使う"というドキュメントもしくはCOMの書籍を見てください．なお，以下の項目がIUnknownインターフェースとなっていますが，サービスコンポーネントのメソッドはIDispachに実装するルールになっているので忘れずに変更してください．

```
[
     uuid(9a966848-e868-4bf7-a6e5-
                              68afe72f60c6),
     oleautomation,
     pointer_default(unique)
]
interface IUPnPService_DimmingService_SCPD
                                   : IUnknown {
               ↓
interface IUPnPService_DimmingService_SCPD
                                   : IDispach {
```

5) デバイスコンポーネントの登録

デバイスホストのIUPnPRegistrarインターフェースを使用してデバイスコンポーネントをデバイスホストに登録します．登録するとデバイスホストはUDNを生成し，完全なデバイスディスクリプションを構築し，ネットワークへデバイスをアドバタイズ(アナウンス)します(このときにサービスディスクリプションのアクションに対するディスパッチルーチンの呼び出しテーブルを構築)．その後，デバイスホストはデバイスコンポーネントを初期化します．じつは，デバイスコンポーネントを登録する方法が三つほどあります．

● IUPnPRegistrar::RegisterDevice

この方法では，デバイスコンポーネントを登録し，アドバタイズまで行われますが，その先には進みません．コントロールポイントからコントロール，イベントの登録などのリクエストがあったときに初めてデバイスコンポーネントをロードし，初期化が行われます．また一度登録すれば，PCが再起動するたびにデバイスがアドバタイズされます．そのため絶えずデバイスコンポーネントを起動するためには良い方法ですが，起動したときに必ずデバイス(デバイスコンポーネントではなく，それがコントロールするリソース，ハードウェアも含む)がきちんと起動していなければなりません．そうでないと，コントロールポイントはデバイスを検出しているのに動作しないという状況に陥ります．

● IUPnPRegistrar::RegisterRunningDevice

この方法は，ユーザーがアプリケーションを起動し，その中でデバイスコンポーネントを登録します．アプリケーションがリソースの状態を確認できるので，準備ができたところでこのメソッドを実行することができます．ですから，デバイスホストはこのメソッドが実行されるとすぐにデバイスコンポーネントをロード，初期化します．ただし，アプリケーションで実行するものなので，PCが再起動するたびに実行されることはありません．

● IUPnPRegistrar::RegisterDeviceProvider

この方法は，IUPnPDeviceProviderというCOMコンポーネントを作りデバイスホストに登録します．登録したとき，およびPCが再起動したときにデバイスホストがIUPnPDeviceProvider::Startメソッドを起動します．このメソッド中でリソースをチェックし準備ができたら，IUPnPRegistrar::RegisterRunningDeviceメソッドを実行し，デバイスコンポーネントを起動します．PCがシャットダウンするときには，デバイスホストはIUPnPDeviceProvider::Stopを実行するので，各デバイスコンポーネントに関する処理を終了します．

なおUDNでは，デバイスホストによりデバイスコンポーネントが登録されるごとに生成されます．それは，そのたびごとにユニークなものです．そのための，同時に異なったコンピュータおよび同じコンピュータに同じデバイスコンポーネントを

〔図12〕utl2idl.exeの実行結果

```
C:¥Program Files¥Microsoft SDK¥Samples¥netds¥upnp¥RegisterDevice>utl2idl  DimmingService_SCPD.xml
XML document loaded successfully
        Translation Done. Check DimmingService_SCPD.idl file
```

登録できます．

サンプルコードの説明

サンプルコードを説明します．まず，**図13**を見てください．デバイスコンポーネントUPNPSampleDimmerDeviceをデバイスホストに登録したあと，コントロールポイントがPowerOnリクエストしたときの例を説明します．

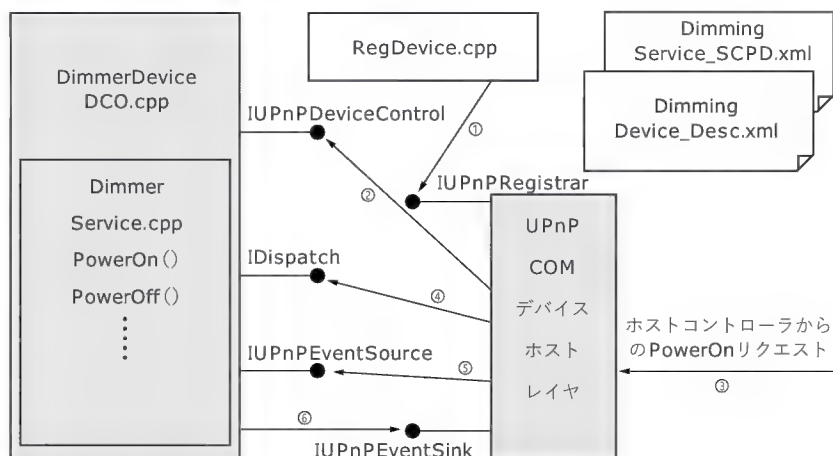
〔図13〕サンプルコードの動作

① RegDevice.cppからUPNPSampleDimmerDevice(デバイスコンポーネント)がIUPnPRegistrar::RegisterRunningDeviceによって登録されます(**リスト12**)．このときデバイスディスクリプションの内容をデバイスホストに渡すため，Dimmer Device-Desc.xmlをRegDevice内で読み込み，ストリング(desDoc)を生成しています．次にUPNPSample DimmerDeviceのインスタンスを生成し，pReg->RegisterRunningDeviceによって，IUnkownインターフェースをデバイスホストへ渡します．これによりデバイスホストは，UPNPSampleDimmerDeviceを認識し，IUPnPDeviceControlをアクセスしていきます．このときデバイスホストは，各ディスクリプションを元にネットワークへアドバタイズします．

② IUPnPRegistrar::RegisterRunningDeviceメソッドでの登録だったので，デバイスホストはUPNPSample DimmerDevice(DimmerDeviceDCO.cpp)のIUPnPDeviceControl::Initializeメソッドを実行します(**リスト13**)．初期化の中でサービスコンポーネントを生成し，IDispatchインターフェースを調べます．ちなみにUPNPDimmerDeviceは，IUPnPDeviceControlインターフェースとして継承されています．

デバイスホストはすぐにIUPnPDeviceControl::GetServiceObjectメソッドをコールし，IDispatchインターフェースのポインタを取得します(**リスト14**)．デバイスホストがサービスコンポーネントのIDispatchインターフェースをコールできるようになり，初期化が終了します．

③ コントロールポイントからPowerOnというアクションのリクエストがあったとします，ホストデバイスはそれを解析し，UPNPDimmerService(DimmerService.cpp)のIDispatch::PowerOnメソッドをアクセスします．

④ ホストデバイスによりDimmerService.cpp（**リスト15**）のUPNPDimmerService::PowerOnメソッドがコールされます．なお，UPnPDimerServiceは，IDispatchインターフェースに継承されています．この中では，仮想的にpowerという変数にフラグを立てています．この中では，esEventingManager->OnStateChanged(1, rgdispidChanges);という処理に注目してください．これはイベント処理の説明で登場します．

〔リスト12〕RegDevice.cpp，195行目～

```
hr = CoCreateInstance(
        CLSID_UPNPSampleDimmerDevice,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_IUnknown,
        (LPVOID *) &punk
        );
  :
  :
hr = CoCreateInstance(
        CLSID_UPnPRegistrar,
        NULL,
        CLSCTX_LOCAL_SERVER,
        IID_IUPnPRegistrar,
        (LPVOID *) &pReg
        );
  :
  :
hr = pReg->RegisterRunningDevice(desDoc, punk, initBSTR,
                             resourcePathBSTR, lifeTime, &DeviceID);
```

〔リスト13〕DimmerDeviceDCO.cpp，81行目～

```
STDMETHODIMP  UPNPDimmerDevice::Initialize(BSTR bstrXMLDesc,
                              BSTR deviceID,BSTR bstrInitString)
{
  :
  :
//サービスコンポーネントのIDispatchインターフェースのポインタ
hr = pUPNPDimmerService->QueryInterface(IID_IDispatch,(LPVOID *)
                                                    &pDisPtrs[0][0]);
```

〔リスト14〕DimmerDeviceDCO.cpp，171行目～

```
STDMETHODIMP  UPNPDimmerDevice::GetServiceObject(BSTR bstrUDN,
                    BSTRbstrServiceId,IDispatch **ppdispService)
{
  :
  :
//Initializeメソッドで調べたIDispatchインターフェースの
//ポインタをセット
    *ppdispService = pDisPtrs[foundDevice][foundService];
    return S_OK;
}
```

⑤ 別のコントロールポイントがイベントのサブスクライブ(購読)をすると，デバイスホストはサービスコンポーネント(UPNPDimmerService)のIUPnPEventSourceインター

〔リスト15〕DimmerService.cpp，77行目～

```
HRESULT UPNPDimmerService::PowerOn()
{
   DISPID rgdispidChanges[1];

   EnterCriticalSection(&csSync);

   if (power != VARIANT_TRUE)
   {
      power = VARIANT_TRUE;
      LeaveCriticalSection(&csSync);

      // We have to send an event here indicating that the
                                    value of Power has changed
      if(esEventingManager)
      {
         // Send the event that power = TRUE through
                  OnStateChanged API exposed by IUPnPEventSource
         rgdispidChanges[0]=DISPID_POWER;
         esEventingManager->OnStateChanged(1, rgdispidChanges);
      }
   }
   else
   {
      LeaveCriticalSection(&csSync);
   }
  return S_OK;
}
```

〔リスト16〕DimmerService.cpp，197行目～

```
HRESULT UPNPDimmerService::Advise (IUPnPEventSink
*punkSubscriber)
{
   // We have to get query for the IUPnPEventSink Interface to
   // send events later using the OnStateChanged API

   HRESULT hr = S_OK;

   // Query the pointer passed to the Advise function for the
                                       IUPnPEventSink Interface
   hr = punkSubscriber->QueryInterface(IID_IUPnPEventSink,
        (void **)&esEventingManager);
   if (FAILED(hr))
   {
      OutputDebugString(_TEXT("UPNPDimmerService: Query
                     Interface failed.  Could not get pointer to
                     IUPnPEventSink¥n"));
      esEventingManager = NULL;
      return hr;
   }

   return hr;
}
```

〔写真1〕
(株)エヌ・ティ・ティ・エムイーのBA 8000 Pro

フェースのAdviseメソッドをコールします(**リスト16**).このときホストアダプタは，それ自身がもっているIUPnPEventSinkインターフェースのポインタを渡します．UPnPデバイスアーキテクチャでは，サービスがもっている変数(イベントの対象になっている)の値が変化すると，購読したコントロールポイントへ通知することになっています．

このインターフェースは，サービスコンポーネントがデバイスホストへ変数の値が変わったことを知らせるためのインターフェースです．esEventingManagerにインターフェースのオブジェクトが用意されます．なお，UPNPDimmerServiceはIUPnPEventSourceインターフェースにも継承されています．

⑥ 少し時間を戻します．PowerOnされたときに，PowerOnをリクエストしたものと別のコントロールポイントがすでにイベントの購読をしていたとします．そこでPowerOnを実行されると，④のesEventingManager->OnStateChanged(1, rgdispidChanges);が実行され，値が変化したことをデバイスホストへ知らせます．デバイスホストは変数と変化した値を通知します．

以上がデバイスホストAPIの説明です．なお，説明していない内容も多くあります．とくに登録を解除，終了する処理，セキュリティに関してはふれていないので，ぜひPlatform SDKをご参照ください．

UPnP NATの概要

Windows XPには，もう一つUPnPにかかわるAPIがあります．UPnP NAT APIです．これもCOMでシステムへ実装されています．これは，IGD(ルータ)をコントロールして，ルータの設定されているステータスやNAT越えするためのコントロールを行います．誌面の関係上，簡単ですがサンプルを使って説明します．なお，Platform SDKには，まだサンプルが用意されていません．そのため，筆者がJScriptによって作成しました．

サンプルの動作環境

今回，IGDには，(株)エヌ・ティ・ティ・エムイーのBA 8000 Proを使用しました(**写真1**)．UPnPのIGD Version1.0の規格に忠実に作られているのでこれを選択しました．UPnP Implementers Corporation(http://www.upnp-ic.org/default.asp)でロゴを取得して，テストをパスしたものがよかったのですが，日本では，それほど数がなく今回は選択できませんでした．この製品の製造元は，プラネックスコミュニケーションズ(株)です．すでにUICのメンバになっているので，UPnPロゴが着いた製品が発売されるのではないかと思います．

〔リスト17〕新しいポートをマッピングするサンプル

```
WScript.Echo ("開始");
var str = Main();
WScript.Echo (str);
WScript.Echo ("終了");

function Main()
{
    var str = "";
    var objNAT = new ActiveXObject ("HNetCfg.NATUPnP.1");
    if (objNAT == null)
        return ("NATUPnP コンポーネントの生成に失敗した. ");

    var objSPMC = objNAT.StaticPortMappingCollection;
    if (objSPMC == null)
        return ("IGDが見つからなかった. ");

    Add (objSPMC, 2048);

    return ("成功");
}

function Add (objSPMC, iIndex)
{
    objSPMC.Add (iIndex,
                 "TCP",
                 iIndex,
                "MSKK-TEST01" ,
                 true,
                 "test__"+iIndex);
}
```

UPnP NATサンプル

二つサンプルを用意しました．一つ目がAddメソッドを使用して新しいポートをマッピングするものです(**リスト17**)．NATUPnPでインスタンスを生成し，StaticPortMappingCollectionインターフェースにアクセスします．次にAddメソッドで2048のポートをTCPで割り当てます．“MSKK-TEST01”は，クライアント(IGDから見て)のコンピュータ名です．IPアドレスを“197.99.60.71”というようにダイレクトに指定してもかまいません．

次は，このコレクションをすべて収集して表示するプログラムです(**リスト18**)．コレクションアイテムをEnumeratorで列挙し，表示します．

このプログラムを実行した結果を示します．**図14**の最後の行を見てください．Addメソッドで追加した分のポートを表示しています．

〔リスト18〕コレクションをすべて収集して表示するサンプル

```
WScript.Echo ("開始");
var str = Main();
WScript.Echo (str);
WScript.Echo ("終了");

function Main()
{
    var str = "";
    var objNAT = new ActiveXObject ("HNetCfg.NATUPnP.1");
    if (objNAT == null)
        return ("NATUPnP コンポーネントの生成に失敗した. ");

    var objSPMC = objNAT.StaticPortMappingCollection;
    if (objSPMC == null)
        return ("IGDが見つからなかった. ");

    // ポートマッピングデータをすべて表示
    Dump (objSPMC);
    return ("成功");
}

function Dump (objSPMC)
{
    var str = "";
    var objPortEnum = new Enumerator (objSPMC);
    objPortEnum.moveFirst();

    var count = objSPMC.Count;
    for (var i=0; i<count; i++) {
        var objSPM = objPortEnum.item();

        str +=   "Description: "        + objSPM.Description;
        str += ", ExternalPort: "       + objSPM.ExternalPort;
        str += ", Protocol: "           + objSPM.Protocol;
        str += ", InternalPort: "       + objSPM.InternalPort;
        str += ", InternalClient: "     + objSPM.InternalClient;
        str += ", Enabled: "            + objSPM.Enabled;
        str += ", ExternalIPAddress : " + objSPM.ExternalIPAddress ;
        str += "¥n";

        objPortEnum.moveNext();
//      if (objPortEnum.atEnd())
//          break;
    }
    WScript.Echo (str);

    return;
}
```

〔図14〕サンプルの実行結果

```
Windows Script Host
Description: MSMSGS (192.168.1.20:14435) 57337 TCP, ExternalPort: 57337, Protocol: TCP, InternalPort: 14435, InternalClient: 192.168.1.20, Enabled: true, ExternalIPAddress: 157.60.99.51
Description: MSMSGS (192.168.1.20:9429) 27109 UDP, ExternalPort: 27109, Protocol: UDP, InternalPort: 9429, InternalClient: 192.168.1.20, Enabled: true, ExternalIPAddress: 157.60.99.51
Description: test_2048, ExternalPort: 2048, Protocol: TCP, InternalPort: 2048, InternalClient: 128.19.33.64, Enabled: true, ExternalIPAddress: 157.60.99.51
OK
```

まとめ

今回Windows XPのUPnP APIを紹介しましたが，Windows XP Embedded，Windows CE .NET Version 4.0以降でも基本的な構造は同じです．参考になると思います．なお，UPnP NATの説明が誌面の関係上，簡単になってしまいました．じつはJScriptでは，StaticPortMappingインターフェースをアクセスできません．Addメソッドの7番目のパラメータにStaticPortMappingのポインタが入ってくるのですが，JScriptでは，アクセスすることができません．実際のプログラムを作成するときにはC++を選択することをおすすめします．

また，機会があればC++でのケースも紹介したいと思います．

参考文献

1) Universal Plug and Play Device Architecture V1.0, http://www.upnp.org/download/UPnPDA10_20000613.htm
2) Windows XPのUniversal Plug and Play, http://www.microsoft.com/japan/windowsxp/pro/techinfo/planning/upnp/UniversalPlugandPlayinWindowsXP.doc
3) Microsoft Windows Software Development Kit February 2003

ながお・やすし　マイクロソフト(株)

● 抽象芸術は何を語る？

小学生だった次男が，学校で描いた絵を得意そうにもって帰ってきたことがある．それがどんなものだったか，今でも鮮明に憶えている．真っ黒い紙の上にカラフルな丸があちこちに描いてある．さらに材質不明の暗紫色の紐が多数張ってある．この絵は何だとしつこく聞く筆者に，次男の答えは鈍かったが，絵の下についている紙は，『宝石を守る龍』という不思議なタイトルを「寡黙に」示していた．

暗い絵を書く子供は，精神的に問題があるという話を聞いたことがある．この絵は，間違いなく，思いっきり暗い．心配になって妻にこのことを話したが，気にも留めてくれない．じつは，学校の先生が黒い色が好きで，子供はいつも暗い絵ばかりを描いているという．何だか，少し心配になってきた．

しばらくして学校で展覧会があったので，行ってみることにした．会場となった体育館の照明がかなり落としてあるせいか，たしかに工作も絵画も，みな暗い．しかし，もっと驚いたことがある．作品を見てみると，常人離れした色彩感覚で描かれた架空の魚や架空の鳥とか，非現実的なものばかりだ．工作も，たとえば黒をベースに派手な色使いがされたこの世のものとは思えない生き物が，二つに割れた卵の殻から飛び出していたりする．この生き物には，いちおう象とか，恐竜とか書いてあるのだが，抽象化されすぎて，それと想像できるものはほとんどない．小学生にしてすでに現実離れしていることに価値観を見出しているとしか思いようがない．

抽象アートの理解能力が完全に欠如している筆者には，どの作品も理解のおよぶところではなかったが，ある種のカルチャーショックを憶える一方で，こんな抽象芸術を教えてどうするのだという怒りとも疑問ともつかぬ思いが湧きあがってきた．

あとで子供に聞いたところ，これらの絵や工作の製作は最高に楽しかったと笑って話してくれた．この明るい答えは，筆者にある程度の安堵感をもたらしてくれはしたのだが……．

その後，気になって，インターネットで小学校の絵画をいろいろと調べてみた．小学校ではホームページをもっているところも多く，ほとんどが児童の絵画を公開している．しかし，そこで見かける絵画は，やはり水彩による人物画，風景画が圧倒的に多い．これを見ていると，小学生らしく，ほのぼのとしていて心が和んだりする．なんだ，全然違うではないか……！

最近，ベテランの小学校の先生と友人になることができた．その先生は，もう四十年近く小学校の先生をやってきた人生の大先輩でもある．性格的にも非常に真面目で，信頼できそうな人だ．そこで，気になっていた，件の話を聞いてみることにした．先生は笑いながら，しかし真剣に答えてくれた．

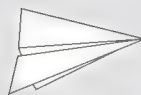
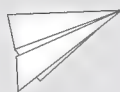
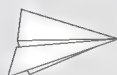
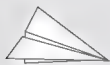
● 最近の子供の気性

現在，都心の小学校の図工では，抽象画や抽象工作しかやらないことが多い．それには二つの理由がある．一つは，子供の性格からくる問題だ．最近の子供は根気がない．面白くなかったり，人からあれこれいわれたりすると，すぐ止めてしまう．たとえ，やりだしたとしても無表情で，少しも楽しそうではない．しかし，現実とはまったくかけ離れた抽象画や，抽象工作ならば，想像力を掻きたてながらとても楽しそうに作ってくれることが多い．もちろん，失敗したって気にもならない．できあがってから，これはこういうものだ，と勝手な題をつければいいのだから．こんなとき，先生はいっさい口を出さない．口を出すと，児童は怒り出したり，工作を投げ出してしまうからだ．

子供が飽きないようにするため，けっこう珍しい高価な材料を教材会社に手配することも多いらしい．これに応えるため，教材会社がいろいろな素材を提供してくれるという．なるほど．いわれてみれば，展示会では何だかわからない新素材も数多く見かけた．

● 差別を避けるため事なかれ主義に

そして，もう一つの大きな理由は，人物画や風景画だと，上手，下手がはっきりしてしまうということだ．そうなると，差別を生む可能性があり，児童の「いじめ」や「不登校」につながることもある．そんなことになると，すぐに親が出てきて，教育委員会をも巻き込んだ大論争になったりする．学校側としては，こんなことはどうしても避けたいため，事なかれ主義におちいる．

今の学校で重要なのは，児童の差別をしないという考えだ．そのため，いつも児童に気をつかっている．たとえば，学芸会では児童の配役に最大限の気をつかうそうだ．なぜかというと，「うちの子がこんな役では困る」といってくる親がけっこう多いらしい．そこで，学芸会では絶対に主役を作らず，みんなを主役にす

るのだという．昔は学芸会には必ず主役がいた一方で，最後まで何一つ喋らない「石ころA」などという可哀想な役もあったものだ．それも社会の要素の一つ，という教育だったのかもしれない．

運動会の「かけっこ」も，昔は背の順に組を作って競争したものだ．しかし今は違う．走る前に何度も練習して，速い順に組み合わせを作り直すそうだ．つまり当日一緒に走る組のメンバは，足の速さがあまり変わらない．本番でも大きな差がつかないから，どの子も公衆の面前で大きな恥をかくことはないのだ．

こんな状況だから，昔はきびしく生徒を指導した先生方も，最近はあまり子供たちの指導をしないらしい．いや，最近は公式には「指導」という言葉は使ってはいけないのだそうだ．代わりに「支援」というのだそうだ．

いろいろ聞いて感じたのは，学校の先生は，どうも教育以外のところで大きなストレスを感じているらしいということだ．

● ひそかに進行する問題

このようになっている理由は，親の過保護と中学進学が原因だという．少子化のせいで，一人っ子が増え，以前とは桁違いに過保護になっているという．また，都心部の場合，私立への進学を希望する生徒の割合が多い．進学塾での勉強は学校よりはるかに進んでいるわけで，学校では非常にやさしいことを，毎日何時間も教わるわけだ．これでは，精神的にまだまだ未完成な児童たちには，一方的にストレスが溜まってしまう．しかし原因がどこにあるのか，ここで論じるつもりはない．

先生の話は極端な例なのかもしれないが，四十年近くを教育の最前線で働いている先生から聞いた本音でもあり，少なからず大きな衝撃を受けた．

一方で，大きな不満も感じた．先生のいうとおりだったら，学校の先生は，いつも問題が起きないよう，児童や先生の間で気をつかってばかりで，結果的に教育に関しては，大きな手抜きをしているのではないか．都会では児童はみな甘やかされて育っているのではないか．だとすれば，これはゆゆしき問題だ．

● 金の卵

すでに死語となってはいるが，昔は中学を卒業してすぐ働きに出る子供たちのことを「金の卵」と呼んだ．使用者側から見れば，安い人件費で一所懸命働いてくれることもあっただろう．しかし，将来大きな可能性を秘めた人材という意味もあったに

違いない．実際，その金の卵たちは，日本の高度成長をしっかりと支えてくれた．

今の子供は，中学を卒業してすぐ働きに出ることはほとんどないが，日本の将来を支える金の卵であることには変わりない．もし先生の話のとおりならば，その金の卵の光は大きく翳ってしまっているのではないだろうか．

著名なシュタイナーの発達段階理論によれば，人間形成のために，7歳までは「意思」の教育を，14歳までに「感情」の教育を，21歳までに「思考」の教育を行うべきだとしている．だとすれば，都会では人間形成に重要な感情教育の期間を無責任な教育に委ねていることになる．最近話題の「切れる17歳」とも無関係ではないかもしれない．

先生が最後に言っていた言葉に次のようなものがあった．「どうせ児童は何年かすれば，学校を去っていくのだから……」

彼が真面目な分だけ，一層この言葉が気にかかった．

あさひ・しょうすけ　テクニカルライター
イラスト　森　祐子

Engineering Life in

凄腕女性エンジニアリングマネージャ（第二部）

▌今回のゲストのプロフィール
エリン・トゥルーロロス（Erin Turullols）：中国系アメリカ人，南カリフォルニア出身．シリコンバレーの地元にある名門大学，スタンフォード大学電子工学部にて学士号（1994年）と修士号（1996年）を取得．電子工学部では，コンピュータアーキテクチャを専門とする．Hewlett Packardの高速ハイエンドチップ設計専門のラボに所属し，チップセットの開発に携わる．その後，マネジメントに興味をもち，プロジェクトマネージャを務める．2001年にPA-RISC開発グループがインテルに譲渡され，それとともにインテルに入社する．現在，サーバ系プロセッサ開発グループのマネージャを務める．趣味はキックボクシング，テニス，バレーボール，サルサダンスなど．

前回まで：さまざまな仕事ができるという可能性からエンジニアリングを選んだいきさつ．そして，Hewlett Packardで管理職に興味をもちマネージャの道を進んだ話と，実際の管理職の仕事について話を伺った．

☆マネージメントへのトレーニング

トニー さて，エンジニアから管理職になったときに，何か会社側からトレーニングなどはありましたか？

エリン HPでは，さまざまな社内プログラムが用意されていました．印象に残るのは，1週間，泊まりがけで行うマネージャ用のトレーニングプログラムでした．次期マネージャになる人達をHP全体で集めて集中トレーニングをするのです．

トニー HPはシリコンバレーでも社員をたいせつにする会社で有名ですよね．ですから，これは管理職もできるだけ社内から輩出しようとするためのプログラムですね．*HP Way*という本が出るほどですから……．それでIntelではどうでした？

エリン たしかにHPのほうが社内の福利厚生などは手厚いですし，私のいた頃のHPだと人事プログラムも社内から作り出す方向が強いですよね．全体的に社員を大事にする会社というのはたしかだと思います．一方Intelは，まったく違ったタイプの会社で非常にビジネスライクだし，競争の激しい会社だと思います．無駄なことをしないようにするし，とにかく出荷してお金を儲けるということに非常にフォーカスしてますね．

トニー う～ん，やっぱり噂どおりで厳しそうですね．元Intelの上司がいたのですが，何か文句を言うといつも言う口癖があって「Intelはもっと厳しいよ……」でした．

エリン Intelではトレーニングコースというよりは，自分で何とかする……が基本的な方針です．幸い私には，HP時代から付き合いのあるメンター（mentor：良き助言者）がいます．彼女も管理職が長いので，いろいろな相談にのってもらったりします．

トニー 女性エンジニア・上司として，何か特別なアドバイスなどは？

エリン とくにないですね……．工学部で女性は非常に少なかったし，大学院ではもっと少なかったので，それは慣れています．ただ現在の仕事では，やはり経験が豊富で歳も私より上のエンジニアを管理する仕事ですから，彼らの信頼を得るまでの時間と努力が必要です．

☆楽しく仕事をする環境作り

トニー 現在17名いるチームを管理して，大型サーバに入れるデバイスを開発しているそうですが，エリンさん独特の管理方法……まあ，チームの管理方法とかありますか？

エリン 私がいつも心がけているのは，「楽しく仕事をしているか？」です．チームのメンバはそれぞれ向上心の強い人達なので自然と仕事に集中するのですが，暗い雰囲気になるといけないと思っています．仕事を通じて何か新しい発見や学ぶことがあるか？　達成感があるか？　チームの連帯感が出ているか？……これらを問いかけて，楽しい雰囲気にしようとしてます．

トニー なるほど……明るいアップビートな雰囲気をたいせつにされているのですね．具体的な例はありますか？

エリン たとえば，テープアウト間近だとバグの検出が大事になってきます．そこでコンテスト形式にして，その週でもっともバグ検出率が高かった人にGoodie Awardを表彰します．まあ，たいしたプレゼントじゃないのですが，映画の券2枚とか，Chevy'sの食事を2名様招待とか……．でもコンテストになっているところが皆の闘志を掻き立てているようです（笑）．

トニー う～ん，ゲーム感覚で最高点を競い合うみたいですよね．でも皆さん多分非常にプロ意識が強く，向上心や仕事への意欲が強いので，頑張るのがわかるような気がします．

エリン あとは，何かチャンスがあれば手柄を立てた人を皆の前で誉めたりすることを心がけています．しかし，オーバワークの人がかなりいて，そこが気になりますね．

トニー 皆さん夜が遅いのですか？

エリン 開発グループ全体で夕食の手配をしているのですが，これがなかなか好評です．各マネージャが，中華，イタリアン，インド料理，メキシカン，ピザとかまあ皆に人気の食べ物を選びます．午後7:30頃に食事が出るのですが，人気が高くてちょっとでも遅いとイライラする人がいたりします（笑）．

トニー ディナープログラムですね．いろいろな会社でやってますが，運営がなかなか難しいものがあります．例を出すと，私が以前いたスタートアップでは，サラトガ（シリコンバレーの高級住宅地の一つ）の遠い所にある高級ケータリングを使っていましたが，とても贅沢なメニューでした．しかし，その会社の運営状況が悪化して，士気が低くなっていくにつれてこのディナープログラムを乱用する人達が増えていきました．8時頃に食事が出るのですが，6時ごろに一度家に戻ってくつろいでから友達とか家族を連れて美味しいディナーを食べに戻ってくる人達とかね（苦笑）．

エリン 現在の職場ではそこまで酷い話はないのですが，たし

Silicon Valley H. Tony Chin

対談編

かに賛否両論ですね．仕事をしてもらうためのワイロのようにも見えますし……バランスがたいせつだと痛感してます．

☆プライベートと仕事時間のバランス……厳しく管理する！

トニー 仕事が忙しいのはもちろんですが，エリンさんはいろいろとプライベートライフでもアクティブですよね．仕事とプライベートのバランスはどう取ってますか？

エリン スケジュールはいつもぎっしりです．私のMS Outlookを見ればすぐわかりますよ……凄いですから（笑）．自分の時間に関しては非常に厳しく管理してます．プライベート面では，自分の趣味や夫との時間をたいせつにしたいので，いろいろと工夫を凝らしてます．たとえば，私の夫もハイテク企業に勤めているのですが，彼は朝がゆっくりなほうなので，私は彼が寝ている間に早起きして出社します．それで午前7:30ぐらいには出社していますが，ほとんどのスタッフは午前10:00すぎに来るため，それまで一人なので非常に集中できるのです．また，お昼の時間もデスクで仕事を続けます．あまり社交的ではないのですが，この一人の時間が非常に貴重なのです．平日の夜はジムに行くかバレーボールの日があり，毎日予定が入ってますのでそれに合わせて会社を出ます．運動した後また戻って仕事をする日もありますが，平均して午後7:00には仕事を切り上げています．ジムで教える日だけは午後5:00に出ます．そして土日は夫婦の時間ということで，予定を入れないようにしています．夫のほうも平日の夜は，ウェイトトレーニングをしにジムに行ったり，インドア・ロッククライミングに通ったりして別行動が多いです．

トニー う～ん，なかなか凄いですね．旦那さんが寝ている時間に仕事の時間を作り出すとは……．CPUのサイクルスチールですかね（笑）．でも，ジムに行ったり体を動かすのは非常に良いことですよね．私も毎日のメリハリを付けるために体を動かすことを日課としています．ところで，食事とかはどうされてますか？

エリン 最近あまり家で作ってないですね……．テイクアウトを利用したり，会社で食事を済ませたりすることが多いです．土日も夫とテニスをしたりアウトドアをするので，そのときに出先で食べたりします．お料理はもう少し頑張りたい分野です．

☆マネージメントをもう少しきわめたい

トニー 今後の予定は？

エリン マネージメントの分野をもう少しきわめたいと思います．上に行けば行くほど全体像が見える気がするのですが，それに魅力を感じています．エンジニアだと直接お客さんに会ったりすることがなかなかないですし……．私の分野だと，私の上司の上司ぐらいになると会うそうです．後は，マーケティングという分野も面白いと感じています．

トニー スタートアップなどは？

エリン・トゥルーロロス氏

エリン 私はずっと大きな会社に勤めて来たので，スタートアップなどの小さい会社の経験がまったくありません．夫はスタートアップに現在勤めていますが，それを見る限りでは私には向いていないと思います．

トニー それはなぜですか？

エリン 夫の場合は，設計の管理をしたり，実際の設計をしたり，機材をベンダから取り寄せたり……LANのメンテをしたり……何役もこなしていますが，私はインフラがしっかりした環境で集中して仕事ができるほうが良いのです．自分のパソコンが壊れたらすぐ誰かが来てくれる環境みたいな……ちゃんとしたITサポートがいる会社が良いです（笑）．

トニー そういうカップルは多いですよね．どちらかが比較的にリスクの高いスタートアップでもう一人が安定した大企業とか．

エリン そこまで意図的ではないのですが，結果的にそうなりましたね．ただスタートアップでは，会社内の政治的なことが少ないのは良いと思います．つまり，仕事が山ほどあるので誰も政治的になる必要はない……．今の職場では，縄張り意識が強い人などもいますから，そういうのはやはり疲れますね．

トニー 人数が多くなるとどうしても何かそういう摩擦が起こるのはどこにでもあるようですね．しかしエリンさんは，人と仕事をするのが良いみたいですね？　いずれはCEOになったり？　HPのフィオリーナさんみたいに？

エリン そこまでいけると凄いと思うのですが，おっしゃるとおりたしかに管理職でも，自分のスタッフが成長したり何か新しい発見があるところを見ていると，とても達成感を感じます．

対談を終えて：

エリン氏は，筆者の通うジムで出会った人だ．こちらのジムのインストラクターは本業をもっていて，趣味でジムのインストラクターをやっている人がほとんどだ．さまざまな本業をもっているインストラクターが多いが，Intelで設計のマネージャをやっていると聞いて少しびっくりした．自分のキャリアの方向を自分で決めたりそれに意欲的に動くという，きわめてアメリカスタイルの展開は非常に興味深かった．

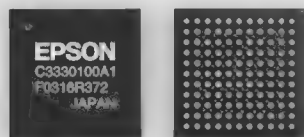
トニー・チン　htchin@attglobal.net　WinHawk Consulting

HARD WARE

●携帯機器向け32ビットRISCプロセッサ

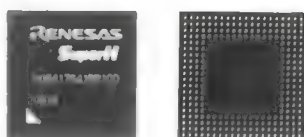
S1C33L11

- 同社の表示コントロールLSI「Mobile Graphics Engine」を内蔵し，携帯電話用のメイン/サブのデュアルLCDインターフェース，JPEGコーデック，カメラインターフェースをハードウェアで処理するため，画像処理のCPU負荷を大幅に軽減．
- JPEGコーデックは，JPEGソフトウェア処理と比較して約1.5倍の高速化と低消費電力化を実現．
- USB1.1ペリフェラル，スマートメディア/マルチメディアカードインターフェース，リアルタイム動画処理ソフト「Nancy CODEC」専用のアクセラレータを内蔵しており，PCなどとのデータ通信やスムーズな動画表示を実現．

■ セイコーエプソン（株）
サンプル価格：¥1,650
TEL ： 042-587-5816

●SuperHマイコン

SH7641

- 最大動作周波数100MHzでCPU性能が130MIPS，DSP性能が200MOPSの高処理性能の「SH3-DSP」コアを搭載している．1サイクルでのアクセスが可能な144KバイトのSRAMおよび16Kバイトのキャッシュメモリで，プログラム処理時間の高速化を実現．
- ACインダクションモータなどの制御が可能な多機能タイマユニットやUSB 2.0対応のUSBファンクション，I²Cバスインターフェースなど豊富な周辺機能を搭載しているため，機器の高機能化が可能．
- 外部データバスは外部メモリを直接接続できるため，機器の小型化や低価格化を実現．

■（株）ルネサス テクノロジ
サンプル価格：¥2,200
TEL ： 03-5201-5214

●16ビット1チップマイコン

H8S/2615F

- 車内LAN規格である「CAN」に対応し，フラッシュメモリを内蔵した16ビット1チップマイコン．
- 動作周波数24MHzを実現し，メモリ，タイマ，シリアルなどの周辺機能を最適化することで，コンパクト化を実現．
- A-D変換機能を4チャネル追加し，16チャネルに機能強化．
- ピン配置で周辺機器などと互換があるため，従来システムの高性能化，高機能化が容易に実現できる．
- 低消費電流化を図った設計により，同じ動作周波数で，従来品と比較して約20%の消費電流低減を実現．

■（株）ルネサス テクノロジ
サンプル価格：¥1,300
TEL ： 03-5201-5216

●汎用ミッドレンジ8ビットマイクロコントローラ

ST72324

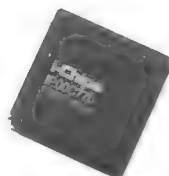
- フラッシュおよびROMに対応した汎用マイクロコントローラで，中/大容量メモリと多ピンパッケージに対応．
- ターゲットは，家電，工業製品，自動車分野．
- インアプリケーションプログラミングが可能なフラッシュプログラムメモリに加え，10ビットA-Dやユーザ設定可能な低電圧検出回路内蔵の組み込みリセット回路などの機能を装備．
- 8K～60KバイトのROM版やフラッシュ版のそれぞれのデバイス間にはピン互換があり，SCI（UART），I²C，SPIなどの標準シリアルインターフェースのほか，PWM対応の16ビットタイマを装備．

■ STマイクロエレクトロニクス（株）
サンプル価格：¥600（500個時）
TEL ： 03-5783-8260　FAX：03-5783-8216

●インタレース-プログレッシブ変換LSI

IP00C770

- インタレースのディジタルビデオ信号をプログレッシブに変換する，ハイビジョン対応の動き適応IP変換LSI．
- 内部10ビット処理を実現し，NTSC～1080iまで広い範囲の画像入力，および1080pまでの画像出力に対応．
- 画像入力ポートはYUV4：2：2 20ビットまたはYUV4：4：4 30ビット，最高で75M画素/s．
- 画像出力ポートは，RGB60ビット/2画素またはYUV4：2：2 40ビット/2画素，最高で150M画素/s．
- 水平，垂直7シンボルの動き検出フィルタを内蔵し，3：2および2：2プルダウン検出機能を備え，フィルムモード処理に対応．

■アイチップス・テクノロジー（株）
価格： 下記へ問い合わせ
TEL ： 06-6492-7277　FAX：06-6492-7388

●SDRAM

EDS2532/2732シリーズ

- ディジタルスチルカメラやディジタルビデオカメラなど，ディジタル民生機器に適する32ビットI/OのSDRAM．
- JEDEC標準のEDS2532シリーズ（4Kロウアドレス）と，動作時の消費電流を20%低減したEDS2732シリーズ（8Kロウアドレス）に，それぞれ動作電圧3.3V，2.5Vの2種類を用意．
- ×16ビット構成の128MビットSDRAM2個搭載と比較し，消費電力が1/2となり，バッテリ駆動のセットに適する．
- 動作周波数166MHz（3.3V），133MHz（2.5V）と高速かつ低電圧での動作が可能．
- セルフリフレッシュ電流は3mA，ローパワー品では1mAを実現．

■ エルピーダメモリ（株）
価格： 下記へ問い合わせ
TEL ： 03-3281-1604

●USB On-The-Goコントロール LSI

S1R72015

- On-The-Go (Supplement to the USB2.0) 1.0版準拠.
- VBUSコントロール用電源機能内蔵による完全1パッケージ化.
- パッケージは，6×6×1.2mmの小型パッケージで提供.
- ハードウェアのファームウェア支援によりCPUの負荷を低減.
- 動作時70mW，サスペンド時0.1mWの低消費電力.
- 3.3V単一電源動作.
- ターゲットは，携帯電話をはじめとする各種携帯機器，ディジタルカメラ，ディジタルビデオカメラ，携帯型ストレージなど.
- 携帯機器間の画像データ，着メロ，アドレス帳のデータ受け渡し，および携帯機器とPC周辺機器間での画像データのプリントアウト，データ保存に適する.

■ セイコーエプソン (株)
サンプル価格：¥1,700
TEL ： 042-587-5816
URL： http://www.epsondevice.com/

●I²Cバスコントローラ

PCA9564

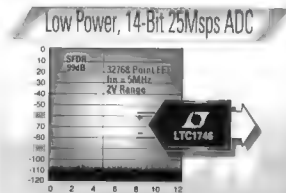
- 400KHz，2.3V～3.6Vの低電圧で動作するパラレル/シリアル変換インターフェース.
- MPU，MCU，DSPと複数のI²CデバイスやSMBusコンポーネントを接続するインターフェースとして適する.
- ビットバンやソフトウェアオーバヘッド増加の必要なしにマイクロコントローラやマイクロプロセッサにI²Cポートの追加が可能.
- スレーブデバイスが同一アドレスを有する，異なる周波数で動作する，あるいはバスロードを分割するなど複数のI²Cバスポートが必要な場合に，MCUまたはMPUにI²Cポートの追加が可能.
- PCF8584を新しい設計に対応させるための低電圧，高周波数の移行バスを提供.
- 8ビットのパラレルデータをシリアルバスデータに変換し，PCボード上での複数トレースの実行を防止.

■ ロイヤル フィリップス エレクトロニクス
価格： 下記へ問い合わせ
URL： http://www.philips.co.jp/semicon/

●A-Dコンバータ

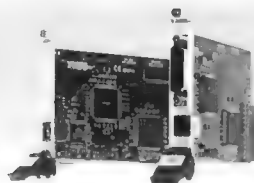
LTC1745/LTC1746

- LTC1745は，380mWの消費電流で72.5dBの*SNR*と96dBの*SFDR*という低ノイズを実現するピンコンパチブルな12ビット，25MspsのA-Dコンバータ.
- LTC1746は，390mWの消費電流で77.5dBの*SNR*と99dBの*SFDR*を達成する14ビット，25MspsのA-Dコンバータ.
- 直線性に優れ，単一5V電源で動作する.
- 入力範囲はどちらのデバイスも±1V～±1.6Vで，アプリケーションに応じてダイナミックレンジを最適化できる.
- 範囲外の入力が発生すると，オーバフローピンの自動インジケータが作動.

■リニアテクノロジ (株)
サンプル価格：LTC1745 ¥1,120～(1,000個時)
LTC1746 ¥1,530～(1,000個時)
TEL ： 03-5226-7291　FAX：03-5226-0268

●ギガビットEthernetモジュール

PXI-8231/PXI-8232

- データ転送速度は，送信では最大500Mバイト/s，受信では最大800Mバイト/sを達成.
- 10Base-Tおよび100Base-TXネットワークとの完全互換性を装備しているため，Ethernetポートはネットワークに応じて動作モードを10Mbps，100Mbps，または1000Mbpsに自動的に切り替えることが可能.
- ストレートケーブルとクロスケーブルを自動的に判別するAuto-MDI機能を備えているため，どちらのタイプのケーブルも利用可能.
- PXI-8232は，IEEE488.2に完全互換.

■ 日本ナショナルインスツルメンツ (株)
価格：¥75,000 (PXI-8231)
¥120,000 (PXI-8232)
TEL ： 03-5472-2970　FAX：03-5472-2977
E-mail：prjapan@ni.com

●ストレージエリアネットワーク専用測定器

1730 ストレージ・エリア・ネットワーク・テスト・システム

- マルチポートファブリックテスト機能により，最大960ポートまでつながるデバイスのエミュレーションが可能なSAN(ストレージエリアネットワーク)専用測定器.
- 通常のトラフィックだけではなく，コントロールプレーンのトラフィックを発生できるコントロールプレーンストレス発生機能を搭載.
- メーカーごとに実装方法が異なることが多い，各スイッチに実装されたネームサーバのコマンドをテスト可能な機能を搭載.
- 故意にエラーを発生させて，そのときのストレージアプリケーションのふるまいを調べることが可能.
- 測定に必要な設定条件や測定開始終了，結果などをわかりやすいインターフェースで一括処理が可能.
- 一度作成した手順を保存し，テストを自動化することも可能.

■ アジレント・テクノロジー (株)
価格：¥5,000,000～
TEL ： 0120-421-345

●オシロスコープ

Infiniium 54833A /Infiniium 54833D

- Infiniium 54833Aは，1GHzの帯域，最大サンプルが4Gsps，アナログ2チャネル，1チャネルあたりのメモリが標準で500Kバイト，最大で16Mバイトを搭載.
- Infiniium 54833Dは，1GHzの帯域，最大サンプルが4Gsps，アナログ2チャネルおよびロジック16チャネル，1チャネルあたりのメモリが標準で2Mバイト，最大で16Mバイトを搭載.
- 従来シリーズ同様，ノブを多用するなど，使いやすいフロントパネルを採用.
- MegaZoomと呼ぶメモリコントローラを採用することで，メモリ長を長くしても十分な画面更新を実現.
- Infiniium 54833Dは，おもにアナログ信号とディジタル信号が混在する機器の開発，デバッグを目的とした機種.
- 既存機種と比較して約40%の低価格化を実現.

■ アジレント・テクノロジー (株)
価格：¥1,694,253～ (Infiniium 54833A)
¥2,118,569～ (Infiniium 54833D)
TEL ： 0120-421-345

■弊誌では新製品に関するニュースリリースを募集しております.
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp　(編集部)

HARD WARE

●スペクトル分析装置

マルチチャネル アナライザー7600

- DSP技術の採用により，回路の小型化，簡素化を実現したスペクトル分析装置.
- アナログ系をなくすことで外来性ノイズによる悪影響を減少.
- パラメータ設定を大幅に簡素化し，身近な設計技術を実現.
- 計測器内部の切り替えスイッチやジャンパを完全になくし，装置の信頼性とより高い保守性を実現.
- 多チャネル制御は，複数の計測系を持つユーザーに優れたコストパフォーマンスを提供.
- 環境測定，材料測定や宇宙，医学，品質管理，基礎研究，教育などスペクトル分析を必要とする分野に適する.

■ セイコー・イージーアンドジー(株)
価格：¥2,000,000～
TEL：03-5645-1777

●ルータ

SEIL/Turbo

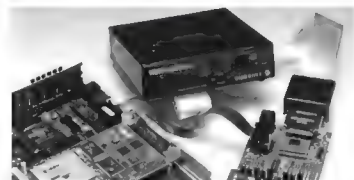
- ギガビットインターフェースを2ポート実装し，280Mbpsのルーティング性能を実現.
- インターネットVPNを実現する高度なIPsec/IKE機能を搭載し，専用ハードウェアによる118MbpsのVPN性能を実現.
- ステートフルパケットインスペクションによるファイアウォール機能を装備.
- コンパクトフラッシュスロット，Mini PCIスロットを装備することで，将来的な拡張性を確保.
- サイズ190×268×45mm，重量1.7kg.
- ファンレスによる壊れにくいハードウェア設計により，高信頼性を実現.

■（株）豊通シスコム
価格：¥780,000
TEL：052-584-8979　FAX：052-584-5587
E-mail：seil@tsyscom.co.jp
URL：http://www.tsyscom.co.jp/seil/

●JTAGデバッグツール

WIND POWER *ICE*

- JTAGServerにより，同一スキャンチェーン上にある複数の異なるCPU，SoC，FPGAおよびCPLDなどのデバイスを最大8デバイスまで1台で同時にデバッグ可能.
- JTAGAccelerator機能により，JTAGアクセスを最大3倍に高速化.
- ホストAPIの公開により，特有の開発環境を構築することができ，ボードテスト用アプリケーションの開発が可能.
- 統合開発環境WIND POWER *IDE*は，1台のホストで同時に複数のデバイスをコントロール，または複数のホストから各デバイスにアクセスできる.
- コンパイラ，エディタ，ISSなどで構成されているため，ハードウェアの完成前にプログラム検証が可能.

■ ウィンドリバー(株)
価格：下記へ問い合わせ
TEL：03-5778-6001

●SoC開発用統合設計環境

Xtensa Xplorer

- 基本設計管理，テンシリカプロセッサコンフィギュレーションツール，ソフトウェア開発ツールのコックピットとして設計を支援する．Xtensaプロセッサ用ビジュアル環境.
- TIE命令により，特定のアプリケーションの性能を最大にするような定義の命令拡張を，プロセッサに対して加えることが可能.
- 異なるプロセッサとTIEコンフィギュレーションは保存され，ターゲットのC/C++ソフトウェアに対してプロファイルを行い，比較が可能.
- パフォーマンスをスプレッド形式の比較チャートに表す，自動グラフ化ツールを用意.
- TIEソースコードエディタ，TIE命令が認識可能なデバッガ，ゲート数見積もりを提供することで，拡張命令に対するハードウェア設計とソフトウェア設計のギャップを埋める.
- スタンダードエディション，プロセッサデベロッパーズエディション，アドバンストエディションの三つの製品を提供.

■ テンシリカ(株)
価格：$5,600（1年間のライセンス料）
TEL：045-477-3373

●FPGA開発キット

PCI開発キット Stratixエディション

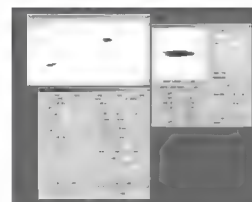
- Stratix FPGAファミリ向けのPCI/PCI-X開発プラットホーム.
- PCI-X 2.0 Mode1システムおよびPCI 2.3デザインを完全サポート.
- SODIMMソケットを介した，ダブルデータレート（DDR）メモリインターフェースを搭載.
- リファレンスデザインが含まれており，購入後すぐにPCIインターフェースとDDRメモリ間のトランザクションを開始できる.
- 3.3Vおよび5Vシステムデザインをサポートする，ユニバーサル（3.3/5V）ショートカードが含まれる.
- Ethernetネットワークへの接続を可能にする，オンボードRJ-45コネクタを搭載.
- Jungo Software Technologiesのドライバ開発キットであるWinDriverをベースにしたWindowsソフトウェアドライバを装備.

■ 日本アルテラ(株)
価格：$1,995
TEL：03-3340-9480　FAX：03-3340-9487

●ラインセンサ欠陥検査装置

WebInspector II TypeB

- ガラス，フィルム，紙，プラスチック，アルミシートなどのシート材の欠陥検査システムの核となる検査部分を，Windows対応PCベースで構築可能.
- 対象物の明欠陥暗欠陥を判別し，欠陥画像を切り出し記録するアプリケーションと，欠陥検出ボードで構成され，毎分150mの処理能力を備える.
- アプリケーションのソースコードが公開されているため，カスタマイズが容易.
- 欠陥検出ボードは，二値化やランレングスなどハードウェアによる前処理機能回路部分を要求仕様に合わせてカスタマイズが可能.
- PCIバス対応製品のほかに，CompactPCIバスに対応した製品も用意.

■（株）アバールデータ
価格：¥728,000
TEL：0427-32-1020　FAX：0427-32-1022

HARD WARE

●CFカードPIOアダプタ

AXC-PI01

- CFカードサイズのパラレル入出力アダプタカードで，PDAなどによるパラレル入出力(14点)を行うことが可能.
- I/F仕様は，CompactFlash仕様R1.4準拠.
- I/O仕様は，Dsub 15ピン 3.3V CMOSレベル対応.
- 内部レジスタ操作により，入出力ピン最大2本を割り込みに割り当て可能.
- 外形は，CompactFlash仕様R1.4準拠のTYPE Ⅰカード.
- ソフト設定により1ピンごとに入出力，プルアップ，プルダウン，オープンドレインの設定が可能.
- 対応OSは，WindowsCE(Pocket PC 2002日本語版).

■ (株)アドテックシステムサイエンス
価格：¥28,000
TEL：045-331-7575　FAX：045-331-7770

●64ビットRISCマイクロプロセッサボード

標準T-Engine TX4956(t101)

- 標準T-Engine仕様に準拠し，米国MIPS社のRISCアーキテクチャを利用した独自開発のプロセッサコア「TX49/H4」を搭載した標準T-Engineボード.
- 最高動作周波数400MHz時において，動作消費電力0.6Wを実現したTMPR4956CXBG-400を採用.
- 命令キャッシュ32Kバイト，データキャッシュ32Kバイトをサポート.
- 128MバイトのRAM，および16Mバイトのフラッシュメモリを搭載.
- シリアルI/Oは，最大38400bpsをサポート.
- サウンドコーデックは，MIC入力1チャネル，Line出力2チャネル，Phone出力1チャネルを搭載.
- eTRON，LCDパネル，タッチパネル，PCMCIA(Type Ⅱ)，USBホスト，拡張ボードなどのインターフェースを装備.
- 拡張バスとして，32ビット/33MHzのPCIバス，および16ビットのローカルバスを用意.

■ 東芝情報システム(株)
価格：下記へお問い合わせ
TEL：044-246-8320

●パソコン計測/制御用機器

Web I/Oシリーズ

- パソコン付属のEthernetインターフェースに直結でき，HUBに接続することでLAN構築，インターネット接続が可能なI/Oシリーズ.
- ユーザーカスタマイズ可能なWebサーバを搭載.
- リアルタイムE-mailアラート機能搭載.
- 10/100Baseの自動検知機能搭載.
- RFC準拠のTCP/IPを搭載.
- 各I/Oには，サンプルアプリケーションソフトを添付.
- 別売で，計測/制御ソフトパック DAC-PAC(V1)を提供.
- 16チャネル，16ビットのアナログ入力，4チャネル，12ビットのアナログ出力を装備.
- リレー接点出力，および4チャネルのRS-232-Cシリアル通信をサポート.

■ (株)ライフトロン
価格：¥50,000～¥100,000
TEL：06-6362-0271　FAX：06-6362-0341
URL：http://www.lifetron.jp/

●開発キット

Oki ML674K/5Kシリーズ対応 RealViewデベロッパーキット

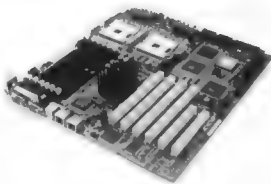
- 沖電気の汎用ARMマイクロコントローラ(MCU)ファミリ専用に設計された，組み込みソフトウェアの開発に必要なツールを統合した開発環境.
- RealViewコード生成ツールと高性能GUIを採用したRealViewデバッガ，およびJTAG ICEで構成される.
- JTAG ICEは，標準的な8MHz JTAGインターフェースをもち，データ転送速度100Kバイト/sのJTAG動作制御を提供.
- 開発したアプリケーションを実際のデバイスで動作確認することが可能な，CPUボードを同梱するパッケージを用意.

■ 沖電気工業(株)
価格：下記へ問い合わせ
TEL：03-5445-6027
E-mail：semi-salesjp@oki.com

●サーバ向けマザーボード

PD-41XE306S2

- Pentium Mプロセッサ，Pentium 4プロセッサおよびXeon(Dual)プロセッサに対応.
- フォームファクタとして標準的なMicroATXおよびSSI-E(Server System Infrastructure Entrylevel)に対応.
- OSは，LinuxまたはWindowsに対応.
- 533MHzのFSBに対応.
- 最大可能搭載メモリは，12Gバイト.
- PCIスロットを6スロット搭載(うちPCI-Xは2スロット).
- ギガビットEthernetポートを装備.
- Ultra320 SCSIを，2チャネル装備.
- Wakeup on LAN機能をサポート.

■ (株)PFU
価格：¥54,000～
TEL：042-788-7726
E-mail：cpsales-g@pfu.fujitsu.com
URL：http://www.pfu.fujitsu.com/prodes/

●LabVIEWベース設計/プロトタイプ作成セット

NI ELVIS

- 工学理論を学習しながら，その論理を電子回路，信号処理，通信，制御システム，機械計測，メカトロニクスなどの領域で実践することが可能な仮想計測器セット.
- LabVIEWベースの仮想計測器，マルチファンクションデータ収録デバイス，およびプロトタイプ作成ボードを備えたカスタム仕様のベンチトップワークステーションで構成.
- オシロスコープ，関数発生器，ディジタルマルチメータ，プログラム可能な電源などに加え，ボーデアナライザ，ダイナミック信号アナライザ，任意波形発生器などを含む一般的な実験機関用計測器を実現.

■ 日本ナショナルインスツルメンツ(株)
価格：¥279,000～
TEL：03-5472-2970　FAX：03-5472-2977
E-mail：prjapan@ni.com

■弊誌では新製品に関するニュースリリースを募集しております.
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp　(編集部)

SOFT WARE

●技術計算ドキュメンテーションソフトウェア

Mathcad 11 日本語版

- 米国マスソフト社が開発した，技術計算/ドキュメント作成ソフトウェア.
- 設計計算や実験データ解析などのさまざまな技術計算を行うだけでなく，計算の過程や前提条件，結果に対する考察などの関連情報を統合した実行計算可能なドキュメントを作成する機能を提供.
- 微分方程式ソルバ機能および関数機能の強化.
- 日本語インライン機能やUndo/Redo機能など，ワークシート編集機能を強化.
- ユーザーインターフェースの日本語/英語切り替えのサポートやオンラインリソースツールバーなどを追加.
- バイナリファイル入出力関数を追加.
- ファイル入出力コンポーネントが，形式の混在したデータを含むデータに対応.
- Excelとのデータ交換機能をサポート.
- リッチテキスト形式ファイルへの変換機能をサポート.

■ 岩通計測（株）
価格： 下記へ問い合わせ
TEL ： 03-5370-5474　FAX：03-5370-5492
E-mail：info-tme@iwatsu.co.jp

●Windows用ソフトウェアRAS製品

INcase

- Windowsパソコンの障害対策を目的としたソフトウェア.
- 監視機能では，パソコンのWindowsの過負荷状態や停止，ハードウェアの異常などを検出する機能で，Windowsとは独立した環境でWindowsやアプリケーションプログラムの動きを監視する.
- 制御機能は，異常を検出したときにあらかじめ決められた手順にしたがって再起動や，データ保護，外部機器の制御などを行う機能で，再起動によってパソコンはリフレッシュされるため，システムの運行が安定する.
- ランプや外部信号，電話回線，ネットワーク，電子メールなどを経由して関係者に異常発生を知らせる通知機能をもつ.
- ネットワーク構成のシステムでは，監視機能，制御機能および通知機能を遠隔で保守する機能を備える.

■（株）マイクロネット
価格： オープン価格
TEL ： 0299-90-1733　FAX：0299-92-8557
URL： http://www.mnc.co.jp/

●XMLネットワーキング製品

XA35 XML Accelerator
XS40 XML Security Gateway

- 米国データパワー社が開発した，XMLネットワーキング製品.
- XA35 XML Acceleratorは，XML処理技術「XML Generation Three（XG3）」を採用し，XMLとXSLTの高速処理が可能.
- XS40 XML Security Gatewayは，XML Webサービスに必要なフィルタリング，暗号化などの各種セキュリティ機能を1台で提供.
- ネットワーク上の最適な箇所にXML専用機器を配置して，問題点を的確に解消する.
- プログラミング不要で，Webサービス標準規格の最新機能を提供.
- Webサービスの課題であるレスポンス，スケーラビリティ，セキュリティ問題を解決.
- サーバ投資，開発工数，管理工数など，Webサービスのアプリケーション開発全体のコストを低減.

■ 東京エレクトロン（株）
価格： ¥9,800,000（XA35）
¥18,200,000（XS40）
TEL ： 03-5561-7195　FAX：03-5561-7413
E-mail：sales_net@kabuki.tel.co.jp
URL： http://www.tel.co.jp/cn/

●フローチャート開発コンポーネント

AddFlow for .NET

- ラッサルテクノロジー社が開発した，ダイヤグラム開発コンポーネント.
- .NET Windows フォームカスタムコントロールで，AddFlow ActiveXバージョンとほぼ同一のフローチャート/ダイヤグラム機能を提供.
- 新機能や自由度を拡大した，単純でパワフルなオブジェクトモデル.
- すべてのコードをC#で書き直すことによって，.NETで提供されるインフラストラクチャの利点が使用可能.
- 100% Managed codeで，ランタイムは無料.
- 画像挿入やメタファイル，インタラクティブモードとプログラムモードをサポート.
- 印刷機能，自動スクロールやズーム機能を搭載.
- 46種類の型のノード，20種類の型のリンクやノードの半透明化をサポート.

■ エクセルソフト（株）
価格： ¥83,800
TEL ： 03-5440-7875　FAX：03-5440-7876
E-mail：xlsoftkk@xlsoft.com
URL： http://www.xlsoft.com/

●アプリケーション翻訳/ローカリゼーションツール

SDLinsight 2003

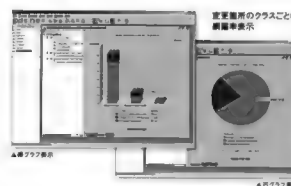
- SDLインターナショナル社が開発した，アプリケーション翻訳，ローカリゼーションツール.
- アプリケーションの翻訳や，ローカリゼーションをVisual Studioライクなビジュアル環境で，効率的に行うことができる.
- プロジェクトの見積もりから翻訳作業，用語集作成，翻訳データベース作成まで総合的なローカリゼーションソリューションを提供.
- SQLサーバを使用した翻訳データベース管理や，バッチプロセスへの統合，.NET環境サポートなどアプリケーション翻訳，ローカリゼーションプロセスの工数削減，プロジェクトの短期化，コスト削減を実現.
- GUI環境でのアプリケーション翻訳，ローカリゼーションをサポート.
- Binary Chopによる国際化問題に対応.
- レバレッジ機能やアライン機能をサポート.

■ エクセルソフト（株）
価格： ¥190,000（Professional）
¥38,000（Translator）
E-mail：xlsoftkk@xlsoft.com
URL： http://www.xlsoft.com/

●ソフトウェア変更検証ツール

Release Rocket Verify for Java

- 米国マッケイブ＆アソシエイツ社が開発した，ソースコードの変更箇所の検証に着目したテストカバレッジ（網羅率）計測ツール.
- ソフトウェアのバージョン間における変更箇所のテスト漏れや，すでにテストが終了した箇所の二重テストを防止し，テストにかかる工数を削減することで，短時間で効率的なテストを実現.
- ネットワークを介したカバレッジ情報を収集するため，エンタープライズ向けやWebベースアプリケーションをはじめとしたJavaによる開発場面での利用が可能.

■（株）エーアイコーポレーション
価格： ¥2,000,000～
TEL ： 03-3493-7981　FAX：03-3493-7993
E-mail：sales@aicp.co.jp
URL： http://www.aicp.co.jp/

●RTOS切り替えツール

OS Changer

- 米国マプソフトテクノロジー社が開発した，特定のRTOS用にカスタマイズされたアプリケーションを，別のRTOSに載せかえることを可能にするソリューション．
- RTOSベースシステム向け開発スイート「eBinder」と統合化されているため，組み合わせて使うことで，pSOSまたはVxWorksから，ロイヤリティフリーのμITRONへの移行が容易に行え，開発期間とコストを大幅に削減できる．
- 単なるラッパではなく，独自の抽象化技術でチューニングされているため，優れたパフォーマンスを実現．
- イーソル製μITRON4.0準拠カーネル「PkKERNELv4」に対応．
- RTOSより下層のプログラムコードの修正を必要とせず，直接RTOS内部のカーネルオブジェクトにアクセスするため，小さいなメモリフットプリントで優れたパフォーマンスを実現．

■ イーソル（株）
価格：下記へ問い合わせ
TEL：03-5301-5325　FAX：03-5376-2538
E-mail：ep-inq@esol.co.jp
URL：http://www.esol.co.jp/embedded/

●組み込みLinux

Lineo uLinux Consumer Electronics Edition

- ディジタル家電や小型携帯情報端末などの，コンシューマ向け機器に適する組み込みLinux．
- Lineo uLinux ELITEに対応するボードサポートパッケージとして提供される．
- 小さいメモリ容量での動作が可能．
- システム起動時間の短縮を実現．
- 省電力に対応．
- ハードリアルタイムを実現する，Realtime機能を搭載．
- メモリ管理機能を充実．
- Kernel2.4.2.0以上をサポート．
- 基本パッケージ群は，RPMとSRPM形式で提供．
- ネットワークは，IPv4/IPv6に対応．
- GNU C/C++，GNU ASM，GNUリンカ，バイナリユーティリティ，デバッガをサポート．

■ リネオ ソリューションズ（株）
価格：下記へ問い合わせ
TEL：03-5322-2856　FAX：03-5322-2858

●組み込み機器開発用コンパイラ

exeGCC Rel3

- GNU Cバージョン3.0に対応し，ITRONのカーネルPkKERNELv4をバンドル．
- PARTNER-JのVLINKに対応．
- リファレンスとして，Solition Platformを使用．
- GNU C++をサポートし，組み込みに適したEC++ライブラリも同時に提供．
- C++対応のPARTNER-ET IIやPARTNER-Jと併用することにより，ユーザーシステムに合わせたC++のデバッグ環境を提供．
- GNU標準のデバッグ情報をサポートし，高速でコンパクトなデバッグ環境を提供．
- PARTNER-ET II/Jが動作しているターゲットでは，ターゲットにファイルシステムがない場合でも，PARTNER-ET II/Jの接続されているパソコンのファイルシステムをそのまま利用可能．
- GNU C/C++を，Windows Me/NT/2000/XPのWIN32に最適化した形でポーティング．
- 新規開発のライブラリには，ANSI C準拠のライブラリと浮動小数点エミュレータライブラリを含む．

■ 京都マイクロコンピュータ（株）
価格：¥148,000
TEL：075-335-1050　FAX：075-335-1051
URL：http://www.kmckk.co.jp/

●CTソリューションGUI開発ツールキット

VBVoice 5.0

- プロネクサス社が開発した，CTソリューション用のGUI開発ツールキット．
- ビジュアル開発環境で，コンポーネントを貼り付ける開発手法で，開発作業の生産性を向上．
- Visual Studio .NET対応で，Visual BASIC V5.0/V6.0以外に，VB.NET/C#での開発が可能．
- VoIP標準仕様のH.323プロトコルをサポートし，従来の開発手法でVoIPをサポートしたCTプログラムの開発が可能．
- 大規模CTソリューションを構築するための分散処理アーキテクチャを導入することで，ネットワーク上の個々の独立したプログラムを統合．
- Dialogic IP-LinkやAculab VoIPなどの音声処理ボードをサポート．
- 電話の自動応答，自動発信，音声メッセージの自動再生，ディジットキーの認識，音声メッセージの録音，FAXの送受信，通話進行の監視と制御などの機能をサポート．

■（株）プロトン
価格：¥248,000
TEL：03-5337-6431　FAX：03-5337-6130
E-mail：cti@sb.proton.co.jp

●品質向上支援ツール

COReTOOL PG-Relief

- 開発工程の早い段階でプログラム品質を高め，ソフトウェア開発のトータルコストの削減を実現．
- C/C++ソースプログラムを静的解析し，演算子の優先順位の誤りや論理的誤りなど障害の原因となる欠陥，移植性や保守性を低下させる記述，性能を劣化させるような記述を指摘．
- 約500ページに及ぶ指摘の解説書をヘルプ化しており，指摘メッセージから該当するソース記述，指摘の意味や対処方法までを表示．
- テスト品質を下げ，プログラムの理解を妨げ，プログラムの保守性を低下させる複雑なプログラムを計測データとして表示する「複雑さ計測機能」を搭載．
- プロジェクトに登録されたすべてのソースプログラムの解析データを集計してファイルに出力でき，プロジェクト全体の品質管理に活用可能．

■（株）富士通インフォソフトテクノロジ
価格：¥1,200,000～
TEL：0120-120-112　FAX：054-202-3121
E-mail：info@ist.fujitsu.com

●組み込みソリューション

GR-XCBASE

- 「GR-XFUNC」「GR-XCTL」「GR-XCOM」の3種類の基盤機能を提供する，携帯，家電機器向けの組み込みソリューション製品．
- 「GR-XFUNC」は，携帯，家電機器の機能拡張/連携基盤で，拡張機能登録，拡張メニュー表示，USB機能拡張モジュールによる機能拡張，インターネット連携による機能拡張などを提供．
- 「GR-XCTL」は，携帯，家電機器の制御定義/管理基盤で，XMLベースによる新制御機能定義，携帯電話などの装置からの機器制御，制御トランザクション管理などを提供．
- 「GR-XCOM」は，携帯，家電機器のWeb/IP通信基盤で，ホームゲートウェイ経由の軽量Web/IP通信，携帯/家電機器の構成制御，アクセス保護，USB機能拡張モジュール通信などを提供．
- 各基盤機能は，単独でも利用可能で他の製品と組み合わせた，利用可能なソリューションとしてさまざまな応用が可能．

■（株）グレープシステム
価格：下記へ問い合わせ
TEL：045-222-3754　FAX：045-222-3759
E-mail：info@solutions.grape.co.jp
URL：http://www.grape.co.jp/

■**弊誌では新製品に関するニュースリリースを募集しております．**
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp　（編集部）

ハッカーの常識的見聞録 33

広畑由紀夫

今月の常識

Intel875P+CSAマザーが登場する！

☆ 前回は，Intel875Pマザーを購入し，実際にそのパフォーマンスを確かめてみましたが，Communication Streaming Architecture(CSA)にふれることはできませんでした．今回は，みなさんが本稿を読んでいる頃には目にできるかもしれないCSAについてふれておきます．

CSA自身はすでにIntel875Pに実装されており，この仕様にしたがっているネットワークチップは6月初旬現在までの発表ではIntel製品の「82547EI」などに限定されます．さて，CSAとはどのようにして高速化をめざしているのか，簡単におさらいしてみます．

● ノースブリッジとCSA

CSA自身は，ノースブリッジに直接接続されるストリーム通信用ポートです．従来，ネットワークカードなどはPCIバスを使用し，AGPの登場前はノースブリッジの機能としておもにCPU-メモリ間転送を行っていました．

その後，AGPの登場でグラフィクスこそノースブリッジで行われるようになったものの，まだまだサウスブリッジ-ノースブリッジ間のボトルネックは解消されそうにないといえます．CSAはAGPと同様にノースブリッジに実装され，サウスブリッジを使用せずに通信を行うために実装されました．

現在，CSAでは双方向データ転送速度が2Gbpsに達し，サウスブリッジを必要としない転送(HDDからの読み込みなど)に関しては，現在のギガビットEthernetに十分なバス転送レートを確保しています．将来的には，さらに高速なバスが実装されるかもしれませんが，PCIを使用せずにバンド幅を確保するため，PCIバスを使用したEthernetコントローラよりも高速かつ，USBなどのサウスブリッジで接続される機器への影響も避けることができるといえそうです．

● サウスブリッジも進化する

CSAで進化したのはノースブリッジだけではありません．サウスブリッジも進歩しているようです．ノースブリッジとの通信速度そのものは従来どおりですが，Ethernetによる通信がPCIコントローラから行われないことを前提に設計(もちろんPCIバスにEthernetコントローラを接続してもよい)することで，より多くの機能を実装できるようになりました．

そうしたことを受けて，ノースブリッジ接続のCSA以外に従来のEthernetチップを接続するための，10M/100M LAN Connect Interface，およびDual Channel Serial ATAへの対応，さらにはIntel RAID Technology対応(ICH5R)なども見受けられます．

● 今年の主流になるのか？

おそらく自作PCなどではPentium4クラスでDDR2対応チップセットが発売開始になるまでは，Intel875P+CSA対応Ethernetがハイエンドクラスとして発売されるのではないかと思われます．筆者が聞いている話では，すでにASUS社などがP4C800 Deluxe(Broadcomを使用．**写真1**)の後継の発売を予定しているとのことです．

自作PCでローエンドに近いところでは採用は少ないと思いますが，コスト以上に性能を重視するユーザー層――とくにブロードバンドネットワーク環境下で大量のオンラインコンテンツを楽しむには適していると思います．もちろん，FFXIなどの非常に高度な3Dグラフィクスとネットワークシステムを混在させて楽しむゲームにはとても良いシステム形態でしょう．また，このCSAを使用したギガビットEthernetを実装したマザーで，Intel875Pのマザーボードとしての完成を見ることができると思います．

● 重要な利点

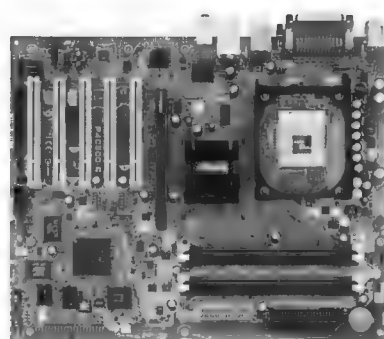
これらの新アーキテクチャの実装による重要な利点は，「アプリケーションレベルではソフトウェア的な仕様変更を行う必要がない」という点でしょう．もちろん，デバイスドライバなどは最適化されたりすることでしょうが，一般的なアプリケーションは何もせずにその性能を享受できるわけです．ハイパースレッディングではアプリケーションのマルチスレッド化などを行わない場合には性能を十分に生かすことが難しかったわけですが，そうした変更をすることもなくパフォーマンスが改善されるのは良いことだと思います．

筆者も先日新調したばかりのP4C800 Deluxeですが，CSA実装マザーが出たらすぐに買い換えて，より快適な操作を味わってみようと思っています．

〔写真1〕P4C800 Deluxe

ひろはた・ゆきお　OpenLab.

海外・国内イベント/セミナー情報

INFORMATION

海外イベント

7/27-31 **SIGGRAPH 2003**
San Diego Convention Center, San Diego, CA, USA
SIGGRAPH
http://www.siggraph.org/s2003/

8/4-7 **Linux World Conference&Expo**
Moscone Convention Center, San Francisco, CA, USA
IDG WORLD EXPO
http://www.linuxworldexpo.com/linuxworldny03/V40/index.cvn

8/17-19 **HOT CHIPS 15**
Stanford Memorial Auditorium, Palo Alto, CA, USA
IEEE
http://www.hotchips.org/

8/20-22 **Hot Interconnects**
Stanford University, Palo Alto, CA, USA
IEEE
http://www.hoti.org/

9/15-18 **Embedded Systems Conference Boston**
Hynes Convention Center Boston, MA, USA
CMP Media Inc.
http://www.cmp.com/eventcal

9/15-18 **TECHXNY/PC EXPO**
Jacob K. Javits Convention Center, NY, USA
CMP Media Inc.
http://www.techxny.com/home.cfm

9/16-18 **COMDEX Canada 2003**
Metro Tronto Convention Centre, Toronto, Ontario, CANADA
Key3Media
http://www.comdex.com/canada/

9/30-10/1 **Embedded Systems Conference Asia**
Lakeshore Hotel, Hsinchu, Taiwan
CMP Media Inc.
http://esconline.com/asia/

国内イベント

8/5-8 **Microsoft Tech ・ Ed&EDC 2003 YOKOHAMA**
パシフィコ横浜(神奈川県横浜市)
マイクロソフト
http://www.event-info.jp/te03/default.htm

8/23-24 **アマチュア無線フェスティバル ハムフェア 2003**
東京国際展示場(東京ビッグサイト，東京都江東区)
(社)日本アマチュア無線連盟
http://www.jarl.or.jp/

9/1-2 **802.11PLANET Conference & Expo Japan 2003**
青山ダイヤモンドホール(東京都渋谷区)
IDGジャパン/米国Jupitermedia社
http://www.idg.co.jp/expo/j802.11/

9/10-12 **自動認識総合展**
東京国際展示場(東京ビッグサイト，東京都江東区)
(社)日本自動認識システム協会
http://www.autoid-expo.com/

9/17-20 **WPC EXPO 2003**
日本コンベンションセンター(幕張メッセ，千葉県千葉市)
日経BP社
http://expo.nikkeibp.co.jp/wpc/ja/

10/7-11 **CEATEC JAPAN 2003**
日本コンベンションセンター(幕張メッセ，千葉県千葉市)
情報通信ネットワーク産業協会，(社)電子情報技術産業協会，(社)日本パーソナルコンピュータソフトウェア協会
http://www.ceatec.com/index.html

10/29-30 **Internet&Mobile 2003**
マイドームおおさか(大阪府大阪市)
日本能率協会
http://www.jma.or.jp/im/

開催日，イベント名，開催地，問い合わせ先の順

セミナー情報

無償ツールによるSystemCデザインのハード/ソフトの検証と合成
開催日時　：7月30日(水)～8月1日(金)
開催場所　：BIZ新宿(東京都新宿区)
受講料　　：198,000円
問い合わせ先：(株)礎デザインオートメーション営業部，☎(03)6762-1471
http://www.ishizue-da.co.jp/

誤り訂正符号の基礎と実際
開催日時　：7月31日(木)
開催場所　：CQ出版セミナールーム
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

暗号化技術の基礎と応用
開催日時　：8月1日(金)
開催場所　：CQ出版セミナールーム
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

C言語によるはじめてのLinuxプログラミング
開催日時　：8月4日(月)～8月5日(火)
開催場所　：DISパソコンスクール(東京都文京区)
受講料　　：92,000円
問い合わせ先：(株)エイチアイICP事業部，☎(03)3719-8155，FAX(03)3793-5109
http://icp.hicorp.co.jp/seminar/linux/clinux.asp

計算力学の基礎コース
開催日時　：8月18日(月)，19日(火)，20日(水)，25日(月)，26日(火)，27日(水)　計6日間
申し込み締め切り日：7月29日(火)
開催場所　：かながわサイエンスパーク(神奈川県川崎市)
受講料　　：72,000円
問い合わせ先：(財)神奈川技術アカデミー，☎(044)819-2033，FAX(044)819-2097
http://home.ksp.or.jp/kast/

～UML2.0リリースで現実となる～MDA(Model Driven Archtecture)技術解説
開催日時　：8月19日(火)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　　：48,000円
問い合わせ先：(株)ソフト・リサーチ・センター，☎(03)5272-6071
http://www.src-j.com/seminar_no/23/23_191.htm

自動車のソフトウェア開発のポイント
開催日時　：8月21日(木)～8月22日(金)
開催場所　：アドバンスト・テクノロジーセンター(東京都千代田区)
受講料　　：60,900円
問い合わせ先：(株)アドバンスト・テクノロジーセンター，☎(03)3518-6441，FAX(03)3518-6147
http://www.at-center.co.jp/pdf/B_2121.pdf

はじめてのTCP/IP
開催日時　：8月23日(土)
開催場所　：CQ出版セミナールーム
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

アジャイル・ソフトウェア開発技法入門解説
開催日時　：8月27日(水)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　　：48,000円
問い合わせ先：(株)ソフト・リサーチ・センター，☎(03)5272-6071
http://www.src-j.com/seminar_no/23/23_141.htm

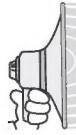
基礎からわかるICタグ/最新技術とシステム応用動向
開催日時　：8月27日(水)
開催場所　：アドバンスト・テクノロジーセンター(東京都千代田区)
受講料　　：60,900円
問い合わせ先：(株)アドバンスト・テクノロジーセンター，☎(03)3518-6441，FAX(03)3518-6147
http://www.at-center.co.jp/pdf/A_1297.pdf

システムコールで学ぶLinux
開催日時　：8月28日(木)
開催場所　：DISパソコンスクール(東京都文京区)
受講料　　：46,000円
問い合わせ先：(株)エイチアイICP事業部，☎(03)3719-8155，FAX：(03)3793-5109
http://icp.hicorp.co.jp/seminar/linux/linuxsystemcall.asp

非接触ICカード/RFタグの基礎
開催日時　：8月28日(木)
開催場所　：中央大学駿河台記念館(東京都千代田区)
受講料　　：52,500円(1口で1社3名まで受講可)
問い合わせ先：(株)トリケップス，☎(03)3294-2547，FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/c030828b.htm

すぐできる「組込みLinux開発」
開催日時　：8月28日(木)～8月29日(金)
開催場所　：(株)ソリトンシステムズ本社7Fセミナールーム
受講料　　：160,000円
問い合わせ先：(株)ソリトンシステムズ トレーニング担当，☎(03)5360-3819
training@soliton.co.jp

ビデオ信号の処理回路技術
開催日時　：9月27日(土)
開催場所　：CQ出版セミナールーム
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

日程はすべて予定です．問い合わせ先にご確認のうえ，お出かけください．

2003年7月号特集「高速バスシステム徹底研究」に関して

▷古いECL信号インターフェースが高速化の流れにのり，LVPECLインターフェースというスタイルで再登場し，注目されているのには驚きです．CPUとメインメモリ間のデータ転送が差動信号インターフェースになると想像すると，非常に頭が痛いです…….（白石 隆）

[編]高速化を追求すれば，いずれそうせざるを得ない状況になりそうです．そうなってから慌てないように，LVDSインターフェースのテストもできるFPGA評価キットなどを使って，差動信号インターフェースを試しながら学ぶなど，今から技術を磨いておく必要がありそうです．

▷ディファレンシャル伝送はノイズに強いというのは知っていましたが，実用例ではUSBやIEEE1394，Ethernetなどケーブルを使った伝送だけだと思っていました．しかしLVDSなどでバスインターフェースなどの実用化もされているということを知りました．とても勉強になりました．（ビギナーズ）

[編]来年にはPCI Expressが立ちあがってきそうですし，PC/AT互換機でPCI Expressの普及がすすめば，数年後にはもっとも一般的な差動信号を採用した拡張バスとなるでしょう．

その他

▷ACPIについてまとまった記事が掲載されていたが，普通にパソコンを利用しているときでもお世話になっている機能ながらあまり詳しくは知らなかったというのが実情で，参考になった．ACPIの管理下におく必要なデバイスを開発しているわけではないが，WindowsのSDKにも情報があるようなので，簡単なツールでも作ろうかと思う．（の）

▷ACPIの記事はナイスタイミングでした．というのも，Windows2000で休止モードから復帰させて放置しておくと，5分程度で勝手に休止モードに戻るという不具合に悩まされています．MSは282208で「不具合ではなく仕様」といっているのですが，そういうものでしょうか？　この問題もいっしょに解説していただけるとありがたいです．（匿名希望）

[編]読者アンケートはがきの「興味の合った記事」でもわかるように，興味をもたれた読者が多かったようです．後編は来月10月号に掲載予定です．ご期待ください．

▷シニアエンジニアの技術草子を興味深く読んだ．現在の日本ではエンタテインメント型のロボットは世界の先端をいっているが，軍事技術が遅れているためトータル的にはけっして高いレベルにあるとはいえない．（天然ヒューマノイド）

▷プロのエンジニアとして，最新技術を無視するわけにもいかず，かといって詳細を追っていく余裕（おもに時間的な面）もない状況で，定期的にある程度つっこんだテーマで検討することができる貴誌に感謝します．（常盤 稔）

特集担当デスクから

☆（プログラムを指しながら）「この辺さあ，ハードウェアで処理すると速いんだよね．この機会にHDLの勉強してみる？」（とあるプログラマ）「うーん，HDLはちょっと……CやC++ならわかるんだけど」

☆全国のプログラマの皆様，お待たせしました．C/C++でハードウェアを記述できる時代がやってきました．手馴れたC/C++をベースとしたシステムレベル記述言語を用いれば，ソフトウェアで行っていた処理をハードウェアで置き換えるほか，ハードウェアもソフトウェアもまとめて設計し，それらを用途に応じて切り分けるハードウェア/ソフトウェア協調設計も可能になります．そのような観点から，第1章から第4章まではハードウェア初学者にも読んでいただけるよう，構成しました．

☆また，すでにハードウェア設計を行っている技術者向けには，第5章のハードウェア/ソフトウェア協調設計が参考になるでしょう．DSPとFPGA，従来はまったく別々の方法で開発していたチップが，協調設計の元に統合されるのです．

☆C/C++のような汎用言語でハードウェアを記述することについて，各種パフォーマンスの観点から疑問視する向きもあるでしょう．しかし，それは時間が解決する問題ではないでしょうか．より大規模に複雑化するシステム設計に対応するためには，C/C++ベースのシステムレベル記述言語という強力な武器が有効でしょう．

☆「ハードウェアは大学で学んだきり」というプログラマの皆様も，この機会にシステム設計を学んでみてはいかがでしょうか．

読者の広場

アンケートの結果

興味のあった記事（2003年7月号で実施）

①プロローグ 高速バスいろいろ
②第6章 USB2.0ハイスピード伝送の実現
③第1章 高速ロジック回路の電気的仕様いろいろ
④組み込みLinuxをとりまく世界（第1回）
⑤家電機器をネットワーク化するアーキテクチャUniversal Plug and Play（UPnP）の全貌（第2回）
⑥第7章 10Gigabit Ethernetの技術動向
⑦Appendix IEEE1394.bの現状
⑧第4章 PCI Express規格の概要
⑨ACPIによるPC/ATの電源管理とコンフィグレーション（前編）
⑩第3章 PC/AT互換機チップセットのデータ転送
⑪第5章 PCI-Xの特徴とプロトコル
⑫第2章 パラレル光モジュールによるデバイス間/ボード間通信の現状
⑬Webサーバ機能をもつEthernet-シリアルコンバータ「XPort」活用技法（前編）
⑭シニアエンジニアの技術草子（弐拾九之段）
⑮開発技術者のためのアセンブラ入門（第19回）
⑯ハッカーの常識的見聞録（第31回）
⑰プログラミングの要（第4回）
⑱XScaleプロセッサ徹底活用研究（第2回）
⑲音楽配信技術の最新動向（第5回）
⑳移り気な情報工学（第33回）
㉑開発環境探訪（第20回）
㉒IPパケットの隙間から（第57回）
㉓Show & News Digest

特集『高速バスシステム徹底研究』についてのアンケートの結果

Q1　今回の特集解説の切り口をどう思われましたか？

①非常におもしろい（30%）
②高速化できた理由が理解できた（8%）
③話としてはおもしろい（38%）
④それぞれの分野を詳しく解説してほしい（15%）
⑤トランジスタの動作などの話は他の雑誌でやってほしい（8%）
⑥その他（0%）

Q2　ふだんあなたが担当されている製品の動作周波数は，だいたいどれくらいですか？

▶CPU/ASIC/FPGAなどデバイス内部

①10MHz程度（8%）　②50MHz程度（25%）
③100MHz程度（33%）④500MHz程度（8%）
⑤1GHz程度（17%）　⑥数GHz以上（8%）

▶デバイス外部/基板間/筐体間

①10MHz程度（33%）　②50MHz程度（25%）
③100MHz程度（17%）　④500MHz程度（8%）　⑤1GHz程度（17%）　⑥数GHz以上（0%）

Q3　興味のあるバス/インターフェースをあげてください（複数回答可）

①Pentiumプロセッサバス（互換チップ含む）（7%）
②PC/AT互換機チップセットバス（7%）
③DDR-SDRAM/ラムバスなどのメモリバス（9%）
④PCI Express（2%）
⑤PCI-X（2%）
⑥PCI（7%）
⑦AGP（2%）
⑧ATA/ATAPI（パラレルATA）（3%）
⑨シリアルATA（5%）
⑩SCSI（2%）
⑪PCカード/CardBus（3%）
⑫USB（18%）
⑬IEEE1394（11%）
⑭Ethernet（16%）
⑮FibreChannel（2%）
⑯その他（2%）DVI

Interface 年間予約購読のお知らせ

Interfaceを確実にお手元にお届けする年間予約購読をご利用ください．

Interface：毎月25日発売
年間予約購読料金：10,800円
※予約購読料金の中には年間の定価合計金額および送料荷造り費用が含まれます．

● **申し込み方法**

お申し込みは，FAXで下記までご通知ください．お申し込みに便利な「年間予約購読申込書」をWeb上でも公開しています（http://www.cqpub.co.jp/hanbai/nenkan/nenkan.htm）．こちらもご利用ください．

お支払い方法は，クレジットカード・現金書留・郵便振替・銀行振込がご利用になれます．

お申し込み受け付け後，請求書を発送いたします．

● **年間予約購読の申し込み先**

CQ出版株式会社 販売局 販売部
TEL：03-5395-2141　FAX：03-5395-2106

読者プレゼント

●応募方法：本誌読者アンケートはがきに必要事項を記入のうえ，2003年8月30日（必着）までにご応募ください．なお当選者の発表は，発送をもってかえさせていただきます．

（1）システム手帳　（1名）

日本ナショナルインスツルメンツ（株）
（http://www.ni.com/jp/）
大きさ：縦24cm×横16cm×厚2.5cm

次号予告

2003年10月号は
8月25日発売です

x86だけではわからない プロセッサの常識

フォンノイマン型/ALU/パイプライン/スーパースカラ/キャッシュ/例外

パソコンはいうに及ばず，情報家電から白物家電まで，現代社会はプロセッサなしには成り立たない．エレクトロニクス技術者として，プロセッサに対する正しい理解は，必修事項といってもよいだろう．

次号では，プロセッサとは何か，パイプライン処理の概念と実際，並列処理の基本とスーパースカラ，キャッシュのメカニズム，割り込みと例外，MMUの仕組み，マルチプロセッサとVLIWなど，プロセッサの基礎から最新CPUの機能まで，プロセッサに関する技術を徹底的に解説する．

また，世の中にはさまざまなアーキテクチャのCPUが登場している．一見するとみな同じようなアーキテクチャに見えるかもしれないが，それはそれぞれのアーキテクチャが，お互いにより良い機能や特徴を取り込んで進化しているからである．このアーキテクチャの変遷を見ることで，より使いやすいプロセッサとはどんなものなのか，プロセッサに関する理解がより深まるだろう．

本特集は規模を拡大して前後編で，10月号と11月号の2号連続の大特集を予定している．

編集後記

■「命は宝」－－通勤の途中，通りすぎた塗装屋さん？の軽自動車の後尾にさりげなく黒文字で記されていたのが目に入った．「命は宝」……何となく頭の中でリピートしながら歩き続け，周囲の緑，緑を目にしたらば，みょーに感動してしまった．生きているって，素晴らしい！　たとえ現実はいろいろシビアかもしれないにしても．　(洋)

■奥さんの携帯が壊れたというので，夫婦揃って機種交換．そして一週間後，自宅のPCが昇天．携帯は3年，PCは7年半で大往生したというわけである．面白いのは，今ほど機種の変更が早い世の中にあって，これだけの年数を何不自由なく済ませていられたことである．もっと魅力的なソフトの開発が急務かも．　(＝IO)

■ついに入手可能になった9.5mm厚/7200rpmの2.5インチHDDを愛機ThinkPadに搭載．速い速いとウハウハ(笑)しながら使ってたら，数日後いきなり電源が落ちる症状が！　何度か電源を入れ直してみると，どうも液晶画面の表示を開始しようとすると電源が落ちる模様．今は外部CRTで使用中……．やっぱり高速HDDのせい？(涙)(M)

■真夏に背広の上着を着ている社員がいる会社はISO14001を剥奪していいんじゃないでしょうか，という暴力的な提案をしてみたい今日この頃．目前の電力不足も気になりますが，日常から環境保護を意識したいですね．まあ突き詰めていけば，“Ultimate ecology is kill yourself”なんですが．それはちょっと．　(み)

■インターフェース誌では「組み込み(Embedded)」という単語を当たり前のように使っているが，組み込みとは何かと聞かれると意外に曖昧である．しかし，厳密な定義はないと思うので，「組み込み機器開発とはCPUを使った専用機器を開発すること」と勝手に考えている．ということは，パソコンは組み込み機器ではない．ちょっと変だが．　(Y)

■ついに出ました!!　ゴキブリ．新築マンションに入居してから5年目．今まで家の中で見かけたことは無かったから殺虫剤も買ってなくてもう大変！　風呂のカビ用洗剤を吹きかけてみたりして大慌てした後，結局は新聞紙でひと叩き！　まったく，どこから入ってきたのか……．とにかく，殺虫剤＆ほう酸団子を買わなくちゃ．　(Y2)

■横浜で写真散歩．掃除用具の置かれた風景をパシャ．後日，写真雑誌を立ち読みしてびっくり．同じ所で写真家・秋山祐徳太子が撮っていた．同じ感性だわ～．以前やはり立ち読みしていた赤瀬川原平の本で香港で私が撮った同じ郵便ポストを発見．にんまり．ところで本は立ち読みしないで買いましょね．　(太陽熱)

■知人の飲み屋が火事で全焼した．再建は早く進み2週間で地鎮祭となった．私も同行したのだがその知人は野球帽を被ったまま地鎮祭に臨もうとする．静寂の中，神主が此方をちらりと見た．私が知人に帽子を脱ぐことを促すと，彼曰く「神主だって被ってるじゃないか」(つかさ)

お知らせ

▶読者の広場
本誌に関するご意見・ご希望などを，綴じ込みのハガキでお寄せください．読者の広場への掲載分には粗品を進呈いたします．なお，掲載に際しては表現の一部を変更させていただくことがありますので，あらかじめご了承ください．

▶投稿歓迎
本誌に投稿をご希望の方は，連絡先(自宅/勤務先)を明記のうえ，テーマ，内容の概要をレポート用紙1～2枚にまとめて「Interface投稿係」までご送付ください．メールでお送りいただいても結構です(送り先はsupportinter@cqpub.co.jpまで)．追って採否をお知らせいたします．なお，採用分には小社規定の原稿料をお支払いいたします．

▶本誌掲載記事についてのご注意

▶コピーサービスのご案内
本誌バックナンバーの掲載記事については，在庫(原則として24か月分)のないものに限りコピーサービスを行っています．コピー体裁は雑誌見開きの，複写機による白黒コピーです．なお，コピーの発送には多少時間がかかる場合があります．
- コピー料金(税込み)
  1ページにつき100円
- 発送手数料(判型に関わらず)
  1～10ページ：100円，11～30ページ：200円，31～50ページ：300円，51～100ページ：400円，101ページ以上：600円
- 送付金額の算出方法
  総ページ数×100円＋発送手数料
- 入金方法
  現金書留か郵便小為替による郵送
- 明記事項
  雑誌名，年月号，記事タイトル，開始ページ，総ページ数
- 宛て先
  〒170-8461　東京都豊島区巣鴨1-14-2
  CQ出版株式会社　コピーサービス係
  (TEL：03-5395-4211，FAX：03-5395-1642)

▶お問い合わせ先のご案内
- 在庫，バックナンバー，年間購読送付先変更に関して
  販売部：03-5395-2141
- 広告に関して
  広告部：03-5395-2133
- 雑誌本文に関して
  編集部：03-5395-2122

記事内容に関するご質問は，返信用封筒を同封して編集部宛てに郵送してくださるようお願いいたします．筆者に回送してお答えいたします．

Interface
©CQ出版(株)　2003　振替　00100-7-10665
2003年9月号　第29巻　第9号(通巻第135号)
2003年9月1日発行(毎月1日発行)
定価は裏表紙に表示してあります
発行人／増田久喜
編集人／相原　洋
編集／大野典宏　村上真紀　山口光樹　小林由美子
デザイン・DTP／クニメディア株式会社
表紙デザイン／株式会社プランニング・ロケッツ
本文イラスト／森　祐子
広告／澤辺　彰　中元正夫　渡部真美
発行所／CQ出版株式会社　〒170-8461　東京都豊島区巣鴨1-14-2
電話／編集部(03)5395-2122　URL http://www.cqpub.co.jp/interface/
広告部(03)5395-2133　インターフェース編集部へのメール
販売部(03)5395-2141　supportinter@cqpub.co.jp
CQ Publishing Co.,Ltd.／1-14-2 Sugamo, Toshima-ku, Tokyo 170-8461, Japan
印刷／クニメディア株式会社　美和印刷株式会社
製本／星野製本株式会社

日本ABC協会加盟誌
(新聞雑誌部数公査機構)　ISSN0387-9569
Printed in Japan